A Scalable Search Engine for the Personal Cloud

Saliha Lallali

NNT : 2016SACLV009

THESE DE DOCTORAT
DE
L'UNIVERSITE PARIS-SACLAY
PREPAREE A
"L'UNIVERSITE DE VERSAILLES SAINT-QUENTIN EN YVELINES"

ECOLE DOCTORALE N°580 STIC
Sciences et technologies de l'information et de la communication

Informatique

Par

Melle Saliha LALLALI

A scalable search engine for the Personal Cloud

Thèse présentée et soutenue à l'Université de Versailles, le 28 janvier 2016 :

Composition du Jury :

M., PACITTI, Esther	Professeur, Université de Montpellier 2	Président
Mr., DU MOUZA, Cédric	Maître de conférences, Conservatoire National des Arts et Métiers	Rapporteur
M., CHABRIDON, Sophie	Maître de conférences, Institut Mines-Télécom/Télécom SudParis	Examinateur
Mr., ILARRI, Sergio	Maître de conférences, Université de Zaragoza, Espagne	Examinatrice
Mr., PUCHERAL, Philippe	Professeur, Université de Versailles Saint-Quentin-en-Yvelines et INRIA Paris-Rocquencourt.	Directeur de thèse
Mr, ANCIAUX, Nicolas	Chargé de recherche, INRIA Paris-Rocquencourt	Invité
Mr., SANDU POPA, Iulian	Maitre de conférences, Université de Versailles Saint-Quentin-en-Yvelines et INRIA.	Invité

UNIVERSITÉ DE VERSAILLES ST-QUENTIN-EN-YVELINES

université PARIS-SACLAY

Titre : Un moteur de recherche scalable pour le Personal Cloud

Mots clés : moteur de recherche, NAND Flash, index inversé, système embarqués, smart object

Le nouveau paradigme du « Cloud Personnel » vise à offrir un espace de stockage et de traitement respectueux de la vie privée, où les données personnelles sont rendues accessibles aux applications autorisées, tout en restant sous le contrôle de l'individu propriétaire des données. Toutefois, rendre le contrôle de la gestion de données à l'utilisateur déporte les problèmes de sécurité au niveau de la plate-forme utilisateur, donc sur le serveur personnel. Ce serveur personnel est en charge d'organiser l'espace personnel de l'utilisateur sous forme d'une base de données orientée documents pour en faciliter la gestion, permettre de croiser des données issus de silos habituellement gérés séparément, et de les protéger contre la perte, le vol et l'utilisation abusive. Ainsi, les opérations de chiffrement/déchiffrement, la gestion des métadonnées (l'indexation et la recherche de documents) et la gestion des contrôles d'accès, sont sous la responsabilité du serveur personnel.

Dans cette thèse, nous proposons une plate-forme de «Cloud Personnel Sécurisé» basé sur un moteur de recherche et de contrôle d'accès embarqué dans un diapositif matériel personnel et sécurisé, relié à la plate-forme de l'utilisateur. Ce type de dispositifs est généralement doté d'une très faible quantité de RAM et d'une grande capacité de stockage Flash, ce qui conduit à des contraintes matérielles contradictoires. Pour faire face à ces contraintes, les moteurs de recherche classique privilégient soit les insertions, soit les requêtes, mais ne peuvent répondre aux deux exigences simultanément. Ainsi, pour constituer une solution réaliste, le « Cloud Personnel Sécurisé » doit s'affranchir de deux difficultés principales. La première difficulté réside dans la conception d'un index inversé sur les documents adapté aux fortes contraintes matérielles du dispositif sécurisé, et permettant à la fois de supporter les mises à jours et l'interrogation de façon efficace sur de grands volumes de données. Deuxièmement, le contrôle d'accès doit être soigneusement intégré dans le moteur de recherche embarqué pour en assurer la sécurité, sans entraver les performances d'interrogation et de mise à jour.

Nous avons implanté notre moteur de recherche et de contrôle d'accès sur une carte de développement représentative de différents dispositifs personnels sécurisés, ainsi que sur un dispositif matériel réel que nous avons fait fabriqué. Nous avons mené des expérimentations approfondies sur de grandes collections de documents réelles et synthétiques. Les résultats expérimentaux ont démontré l'évolutivité de l'approche et sa supériorité par rapport aux méthodes d'état de l'art.

UNIVERSITÉ DE VERSAILLES ST-QUENTIN-EN-YVELINES

université PARIS-SACLAY

Title : A scalable search engine for the Personal Cloud

Keywords : search engine, NAND Flash, inverted index, embedded system, smart object

The emerging Personal Cloud paradigm holds the promise of a Privacy-by-Design storage and computing platform where personal data remains under the individual's control while being shared by valuable applications. However, leaving the data management control to user's hands pushes the security issues to the user's platform, i.e., the Personal server. This Personal server is in charge of organizing the personal dataspace in a document database style to ease its management, to allow crossing data from multiple "local silos" and to protect it against loss, theft and abusive use. Hence, encryption/decryption, metadata management (e.g., indexing and searching the documents) and access control management is under the responsibility of the Personal server.

In this thesis, we propose a Secure Personal Cloud platform relying on a query and access control engine embedded in a tamper resistant hardware device connected to the user's platform. Such devices are generally equipped with extremely low RAM and large Flash storage capacity, which lead to conflicting hardware constraints.

To tackle these constraints, conventional search engines privilege either insertion or query scalability but cannot meet both requirements at the same time. Thus, to become reality, the Secure Personal Cloud has to overpass two main difficulties.The first difficulty lays in the design of an inverted document index capable of tackling the strong hardware constraints of secure devices, and reach update and query scalability at the same time. Second, the access control has to be carefully integrated with the embedded search engine to ensure the security, but without hampering the query and update performance.

We have implemented our engine on a secure token having a hardware configuration representative of tamper resistant devices and have conducted extensive experiments using real, large document collections. The experimental results demonstrate the scalability of the approach and its superiority compared to state of the art methods.

Table of contents

List of figures

List of tables

Chapter I
Introduction

We are witnessing an exponential increase in the acquisition and production of personal data from companies and quantified-self devices or smart objects. At the same time, secure smart objects are emerging and hold the promise of a real breakthrough in the management of personal data. Secure smart objects can embed personal data and/or metadata referencing documents stored encrypted in the Cloud and can manage them under the holder's control. In this thesis, we tackle the problem of a new embedded search engine designed for the personal cloud where the documents in the cloud are encrypted, while their metadata and their related search and access control engine are stored in a secure smart object. Designing such an engine is very challenging due to a combination of conflicting NAND Flash constraints (the block-erase-before-page-rewrite constraint) and embedded system constraints (the scarce RAM space), disqualifying known state of the art solutions.

In this chapter, we first position the context of the Personal Cloud environment. Then, we motivate the Personal Cloud by a use case and expose the difficulty of designing such an engine under the smart object constraints. Finally, we present our main contributions and the outline of the manuscript.

1. Context

We are witnessing an exponential increase in the acquisition of personal data by administrations and companies, and by the individuals themselves e.g., through quantified-self devices [1] [2]. In fine, all this data ends up in servers where it remains organized in silos, each under the control of their data collector. However, the pressure increases to link data between different silos thereby increasing business opportunities. Hence the strategy of major Internet companies that search to horizontally control multiple data silos (domains of interest, leisure, geolocation, health, etc.). However, the PRISM affair[1] has made clear that centralizing and processing all one's data in a single server (or multiple servers controlled by a single actor) introduces

[1] See e.g. the Wikipedia page on PRISM surveillance program "https://en.wikipedia.org/wiki/PRISM_(surveillance_program)"

a major threat on privacy. Moreover, this introduces a monopolistic situation impeding the free development of many valuable applications by others.

To face this situation, the World Economic Forum formulates the need for a data platform that allows individuals to manage under their control the collection, usage and sharing of their data[2]. This is precisely what the Personal Cloud Paradigm aims to provide. Because it gives a tangible source of trust, the home cloud is the most emblematic form of Personal Cloud. It can be thought of as a dedicated box connected to the user's internet gateway, equipped with storage, computing and communication facilities [3], running a personal server and acquiring personal data from multiple sources [4]. This personal server is in charge of organizing the personal dataspace in a document database style to ease its management, to allow crossing data from multiple "local silos" and to protect it against loss, theft and abusive use. Many projects and startups currently investigate this direction like OpenPDS[3], CozyCloud[4], Younity[5], Lima[6], OwnCloud[7], Tonido[8], Seafile[9], SparkleShare[10].

Leaving the data management control to the user's hands pushes the security issues to the user's computing platform as well. Documents can still be stored encrypted in Cloud servers but these servers never get access to the cryptographic keys and their role shrinks to guaranteeing the availability and resiliency of encrypted data. Hence, encryption/decryption, metadata management (e.g., indexing and search) and access control management is under the responsibility of the personal server. This responsibility is paramount considering that all silos of personal data are now grouped in the user's hands.

2. Motivation

To tackle this challenge, we consider the Secure Personal Cloud platform of a typical user, Alice, pictured in Figure 1. It combines a traditional home cloud data system organizing the document dataspace and a secure hardware-based co-server

[2] The World Economic Forum. Rethinking Personal Data: Strengthening Trust. May 2012.
[3] OpenPDS: http://openpds.media.mit.edu/
[4] CozyCloud: https://www.cozy
[5] Younity: http://getyounity.com/
[6] Lima : https://meetlima.com/
[7] OwnCloud : https://owncloud.org/
[8] Tonido : http://www.tonido.com/
[9] SeaFile: http://seafile.com/en/home/
[10] SparkleShare: http://sparkleshare.org/

managing the encryption/decryption of documents and enforcing the access control rules. This secure co-server can be hosted in any of the tamper-resistant devices flourishing today, like Mobile Security Card[11] (produced by Giesecke & Devrient), Personal Portable Security Device[12] (produced by Gemalto and Lexar), Multimedia SIM card or Secure Portable Token [5]. Whatever its commercial name and form factor, a tamper-resistant device, called secure token hereafter, embeds a secure microcontroller (e.g., a smart card chip) linked to a large NAND Flash memory (e.g., a SD card) and can communicate with a host through a USB, Ethernet port or Bluetooth.

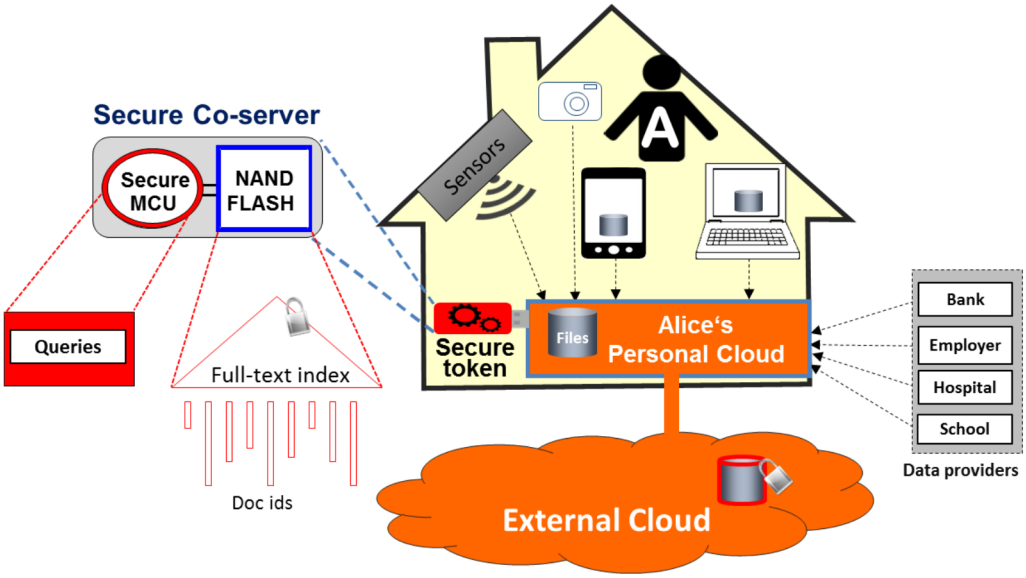


Figure 1: Secure Personal Cloud platform

The documents entering Alice's dataspace can be any form of files (pictures, text files, pdf files, mails, data streams produced by sensors, etc.). To tackle this diversity, we consider a simple keyword search query engine, in the spirit of Google desktop or Spotlight. Terms are extracted from the file content and from metadata describing it (e.g., name, type, date, creator, tags set by the user herself). Since keywords seems the natural way to query Alice's dataspace, it makes sense to let her express access control rules with a similar paradigm, that is through logical expressions on terms associated to documents. Hence, a query issued by a user or an application can be any form of term search expression, with a ranking function (e.g., tf-idf) identifying the top-k most relevant documents. Only the documents granted to the user/application

[11] http://www.gi-de.com/en/products_and_solutions/products/strong_authentication/Mobile-Security-Card-31488.jsp

[12] For example « Smart Guardian » http://cardps.com/product/gemalto-smart-guardian or « Smart Enterprise Guardian » http://cardps.com/product/gemalto-smart-enterprise-guardian

must appear in the query result (e.g., a query issued by the mailer application can only see documents containing the terms email or contact). For example, Bob can search in Alice's dataspace for the *k* most relevant files containing (or tagged with) the terms 'vacation', '2014', 'France', but among these documents can see only those which have been tagged by Alice with the term 'friends'.

3. Problem statement

The question addressed in this thesis is how to make this architecture secure and scalable? To answer information retrieval queries, search engines usually rely on an inverted index containing all relevant terms extracted from the content and metadata of all documents in the collection. This inverted index is thus as sensitive as the documents collection itself and must be protected accordingly (see Figure 1). Unfortunately, unless if Alice is a world-class security expert, her home computing platform cannot reasonably be considered trusted. The unique source of security is provided by the secure token connected to her home platform. This means that the inverted index and the query and access control engine must be embedded in the secure token to avoid any data disclosure at query execution time, except the final result.

Designing such embedded query and access control engine is however challenging due to a combination of severe and conflicting hardware constraints of the secure tokens. Typically, secure tokens are equipped with a tiny RAM and their persistent storage is made of NAND Flash badly adapted to random fine-grain updates. Unfortunately, state-of-the-art indexing techniques either consume a lot of RAM or produce a large quantity of random fine-grain updates. Few pioneer works already considered the problem of embedding a search engine in sensors equipped with Flash storage and tiny RAM [2] [6] [7] [8] [3], but they target small data collections (i.e., hundreds to thousands of files) with a query execution time which remains proportional to the number of indexed documents. By construction these search engines cannot meet insertion performance and query scalability at the same time. Moreover none of them support file deletions and updates, which are mandatory in the Personal Cloud context.

4. Contributions

In this thesis, we make the following contributions:

- We introduce three design principles, namely Write-Once Partitioning, Linear Pipelining and Background Linear Merging, to devise an inverted index capable of tackling the conflicting hardware constraints of secure tokens.
- We define a simple, tag-based access control model to be integrated with the search engine in order to allow the Personal Cloud owner to easily define access control rules and protect her data.
- Based on the above mentioned principles, we propose a novel inverted index structure and related algorithms to support all the basic index operations, i.e., search, insertion, deletion and update while combining them with access control rules.
- We validate our design through an extensive experimentation on a real, representative secure token platform and using real, large datasets and show that query scalability and insertion/deletion/update performance can be met together and combined with high security guarantees. The secure search engine was also integrated in the software platform developed in the context of the ANR KISS project, whose objective is to develop a secure and portable Personal Data Server (see Chapter V).

5. Outline

This thesis is organized as follows:

- Chapter II details the secure tokens' hardware constraints, analyses the state-of-the-art solutions and derives from this analysis a precise problem statement.
- Chapter III clearly states the hardware's conflicting constraints and introduces three design principles, namely Write-Once Partitioning, Linear Pipelining and Background Linear Merging to define a search engine tackling such constraints.
- Chapter IV details the core design of the embedded search engine, as well as the tricky case of documents' deletions, and it explains the integration of the access control in the search engine. We also explain in chapter IV how the proposed solution achieves the RAM bound and full scalability properties.
- Chapter V introduces the scenarios and the objectives of the Secure Personal Cloud, and explains the integration of the proposed search engine in the KISS

project. In chapter V, we also detail the algorithms of the proposed solution and provide additional details of the internal structure of the architecture through a demonstration underlining three results: the full scalability, the RAM bound and the security of the search engine.

- Chapter VI presents an extensive evaluation of the proposed search and access control engine. It introduces the datasets used in the experiments. It discusses the performance of the index maintenance and the querying, and the impact of the access control on the query performance, as well as the impact of the deletion rate on the query performance and the index size. Finally, we compare our search engine performance with the state of the art relevant indexing techniques.
- Chapter VII concludes the thesis and introduces a few promising ideas for future work.

Chapter II
State of the art

With the continuous advances in several domains such as smart objects, the Personal Cloud, sensor networks and Internet of Things, a new technology implementing secure tokens is emerging as well. In this chapter, we present first the hardware configuration and constraints of such devices. Second, we explain the general requirements to implement a search engine that enables full-text queries. Then, we analyze the existing solutions that embed search engines in devices having similar hardware configurations with the secure tokens. We also discuss different indexing techniques that are relevant to the proposed search engine (i.e., B-tree-like indexing in NAND Flash and partitioned indexes). Finally, we will conclude the stat of the art by categorizing the main families of indexing techniques under the hardware constraints of secure tokens.

1. Hardware Constraints of Secure Tokens

Whatever their form factor (e.g. SIM card, USB token, Secure MicroSD) and name (e.g. Mobile Security Card[13] (Giesecke & Devrient), Personal Portable Security Device[14] (Gemalto and Lexar), Multimedia SIM card or Secure Portable Token [9]) (see Figure 2), secure tokens share strong commonalities in terms of data management architecture. Indeed, a large NAND Flash storage is used to persistently store data and indexes, and a microcontroller (MCU) to execute the embedded code, both being connected by a bus.

[13] http://www.gi-de.com/en/products_and_solutions/products/strong_authentication/Mobile-Security-Card-31488.jsp
[14] "Smart Guardian" http://cardps.com/product/gemalto-smart-guardian or "Smart Enterprise Guardian" http://cardps.com/product/gemalto-smart-enterprise-guardian

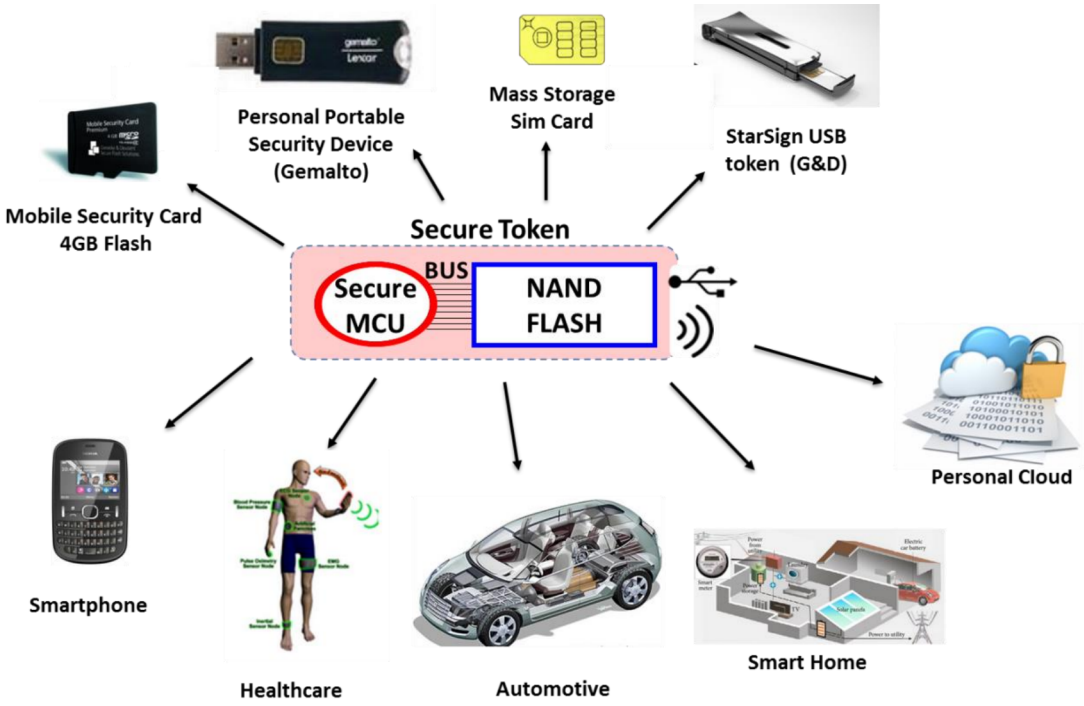


Figure 2: Secure tokens' form factors

The MCUs embedded in secure tokens usually have a low power CPU (e.g, a 32-bit RISC processor, clocked at 168MHz), a tiny RAM (128 KB), about 1MB of NOR[15] Flash memory used to store the code and sensitive data (e.g., cryptographic keys), a cryptographic co-processor and security modules providing the tamper-resistance[16]. The MCU is connected by a bus to a large external NAND Flash storage component (either raw NAND Flash chips or an SD/micro-SD card) and it can interact with the outside world through various standards like wireless communication for sensors (e.g., Bluetooth, RFID, 802.11), USB, Ethernet or Serial (RS-232) for bigger devices.

Secure tokens provide many benefits such as low power, low cost, high portability, and high security, but also exhibit strong hardware constraints inherited from both the MCU and the NAND Flash:

- MCUs have scarce RAM resources (in the order of tens of KB) compared to the gigabytes of stable NAND Flash storage. Indeed, due to its poor density, and

[15] The XIP (eXecute-In-Place) nature of the NOR Flash, allow code execution without loading it into RAM wish lead to reduce RAM consumption, but the extreme writing cost make it less suitable for data storage in addition to its poor density.

[16] Secure chips are protected by hardware module destroying the contents of the chip in case of physical intrusion (detection of abnormal variations voltage, radiation detector, ...) or logical intrusion (interference data bus, firewall protecting the memory,…)

as it competes with other hardware resources on the same silicon die, RAM is and is expected to remain a scarce resource in MCUs, especially when compared to the stable storage with an increasing storage capacity and a scaling density.

- A few MB of NOR Flash memory are also located in the MCU. Therefore, the NOR Flash benefits from tamper resistance of the MCU.
- Unlike the NOR Flash memory, the NAND Flash persistent storage is external to the MCU and does not benefit from the secure MCU's tamper resistance. Hence, the data stored in NAND Flash need to be cryptographically protected against confidentiality and integrity attacks.
- NAND Flash storage exhibits strong limitations, the minimum unit for a read and a write operation is the page (usually 512 Bytes, also called a sector). Pages must be erased before being rewritten but the erase operation must be performed at a block granularity (e.g., 256 pages). Erases are then costly and a block wears out after about 10^4 repeated write/erase cycles. In addition, the pages have to be written sequentially in a block. Therefore, NAND Flash badly supports random writes. We observed this same bad behavior both with raw NAND Flash chips and SD/micro-SD cards. Our own measurements shown in Table 1 corroborate the ones published in [10].

Table 1: Performance in milliseconds of fine grain accesses (of 2KBs) for several micro-SD cards

	Read	Seq. write	Rand. write
Kingston SDHC UltimateXX 8Go-UHS-I	1.1	16	83
Kingston microSDHC 4Go Class 10	1.3	6.3	78
Lexar SDMI4GB-715	0.9	2	30
Silicon Power Secure Digital HC 4Go Class 6	0.9	2	43
Samsung MicroSDHC Plus 8Go Class 10	1.3	2.9	315
Silicon Power microSDHC Class 10 4Go	1.8	3	327
Samsung SDHC Plus 8Go Class 10	1.4	2.9	320
Kingston SDHC 4Go Class 10	1.5	2.5	340
Silicon Power Secure Digital HC 4 Go Class 10	1.1	3.4	660
Sony SF4B4	1.1	2.7	746

However, unlike magnetic disks, a NAND Flash memory contains no moving parts and therefore is not constrained by seek time or rotational latency. This feature makes the random and sequential reads time similar, in addition to its low energy consumption, shock resistance, high density. Given these important benefits, NAND flash memory is currently widely used in embedded systems.

2.Search Engine Requirements

In this subsection, firstly we will introduce IR principles and the inverted index to index documents. A document can be of many types (e.g., text file, image, etc.) and is associated with a set of terms (or keywords) describing its content and weights indicating their respective importance in the document. For text documents, the terms are words composing the document and their weight is their frequency in the document. For images, the terms can be tags, metadata or visterms describing image subparts [11] .

2.1 Introduction to Information Retrieval Systems

An Information Retrieval system (IR) is the set of processes responsible to find relevant documents related to users information from a data set collection [12].

Figure 3 gives the general view of an Information Retrieval system architecture. An IR system is responsible of two main tasks: indexing the document collection and answering user's queries. The indexing process starts by a preprocessing phase to extract the terms from the documents, removes the stop words, converts the terms to their root and casts the letters to lower case. All these steps are discussed in Section 2.2.1. The second phase consists in the construction of the index based on the extracted document terms. Users using the IR system interface formulate queries to extract documents related to their needs. The search engine preprocesses the query in a similar fashion with documents' preprocessing (i.e., do a preprocessing phase of the query terms), interrogates the index, finds the documents relevant to the query, ranks the documents on their score according to their relevance to the query, and finally returns the most relevant documents to the user. The query processing is discussed in more detail in Section 2.3.

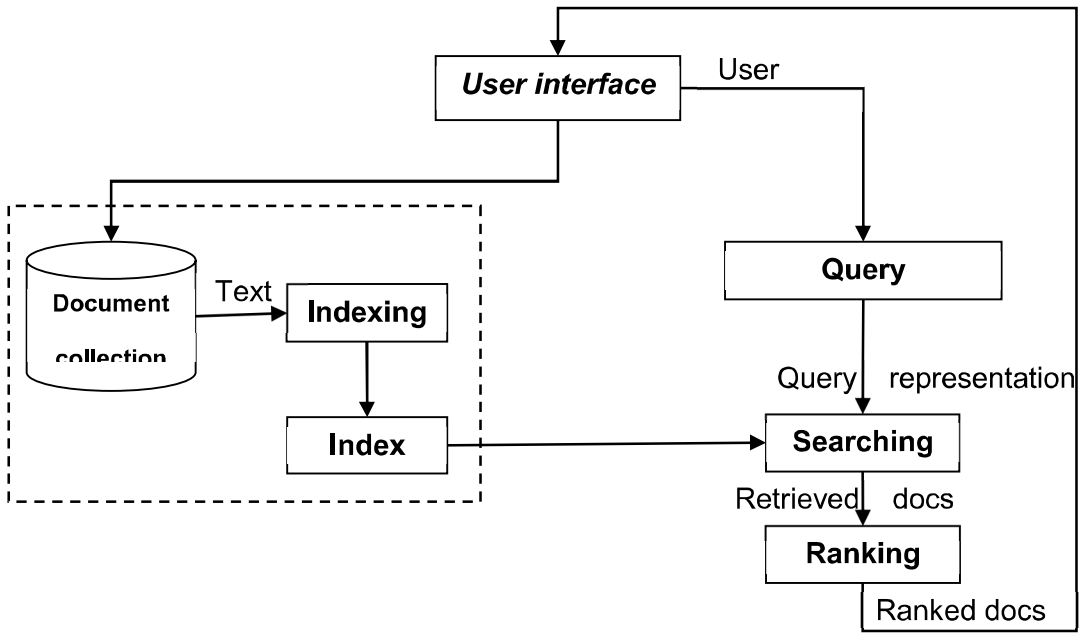


Figure 3: Information retrieval process

2.2 Indexing

2.2.1 Extraction of Index Terms

The preprocessing phase to extract significant words from text documents follows several steps. Typically, document words are extracted and less significant words are discarded, while the remaining words are stemmed to their root as detailed in the following.

2.2.1.1 Tokenization

The first step of document preprocessing is the lexical analysis of the document streams, in which the words are identified and converted to lower cast, the spaces are discarded. Also, generally numbers and punctuation marks are removed.

2.2.1.2 Elimination of stop-words

Words with little meaning (such as the, to, in, there, a, an,…) are filtered out from the document text in the preprocessing phase using either a stop-word list that include all the words to be removed, or by counting each word frequency and eliminating words with a very high frequency (e.g., that exceeds a certain threshold). Removing these words reduces the index size which leads to improve the execution time of queries.

In general, removing the stop-words does not have an impact on the quality of the query results. However, in some specific cases, removing the stop-words can deteriorate the search quality. For a specific query like 'to be or not to be", removing the stop-words can lead to an empty result, but such queries are rare.

2.2.1.3 Stemming

Stemming is a preprocessing technique that consists of removing affixes (suffixes or prefixes) from words and keeping the words' root or stem. For example, the words "accept", "acceptable", "acceptance", "acceptances", "acceptation", "accepted" and "accepting" can all be indexed as "accept".

Stemming decreases the collection size and complexity, and thus improves the time of indexing the documents and the time of query processing. A possible disadvantage of stemming is that for certain search queries it may impact the query relevance, i.e., it may return documents with terms stemmed to the same root which don't satisfy users' needs. But in general, stemming does not affect the search effectiveness. The most common stemming algorithm of words in the English language is the Porter's algorithm [12].

2.2.2 Index Construction

This process is responsible of building a search structure representing the data collection to allow for an efficient query processing. The inverted index [12] is the most common and effective way for IR systems to store document content and be able to perform query search efficiently. Nevertheless, other techniques for text search and indexing exist such as the signature file [13] or Suffix Arrays [14].

2.2.2.1 The structure of the inverted index

In a directory containing a list of documents, each document is identified by a document identifier and to each document is associated a set of terms. The inverted index reverses this structure by storing the collection by terms and by associating to each term the documents' identifiers containing that term. The inverted index structure can be summarizes by: the vocabulary and the set of inverted lists. Figure 4 presents an overview of an inverted index internal structure:

The vocabulary, also called search structure or dictionary (I.S in Figure 4), stores all the distinct terms of the data set (i.e., terms are stemmed, casted to lower case and stop words are removed), and the frequency of the term in the collection F_t, (i.e., the

number of documents containing the term). Each term is associated to the documents containing it by a pointer to its inverted list. Generally a B-tree is constructed to index the vocabulary and accelerate the term search.

The set of inverted lists, also called posting lists (I.L in Figure 4): to each term in the dictionary is associated a set of postings, each of them containing a tuple of (d , $f_{d,t}$) where d is the document identifier that contains the term t and $f_{d,t}$ is the frequency of the term in that document.

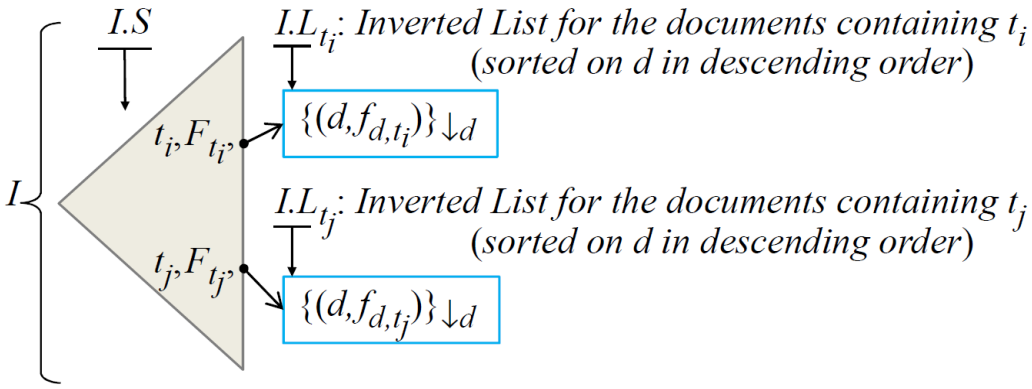


Figure 4: Typical inverted index structure

Figure 5 shows a simple example of an inverted index internal structure for the "proverb data set". As described earlier, the inverted index structure is a combination of two structures: the dictionary part and the set of inverted list parts. Figure 5 shows a *document-level* index where words' positions within the document are omitted. Other information can also be kept in the index such as term type (e.g., author, title,..), links for web indexing, and so on.

To complete the IR structure, other structures can used, such as the documents themselves, a mapping structure between the documents' name in the system and their identifiers in the inverted index, an array recording each document length in the data set by number of words in the document or by bytes. However, such structures are not explored in this thesis.

d_1. A bird in the hand is worth two in the bush

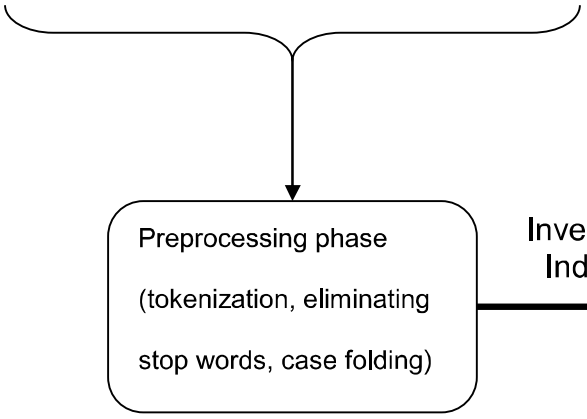
d_2. Birds of a feather flock together

d_3. Better one eye than quite blind

d_4. The early bird catches the worm

d_5. In the kingdom of the blind, the one eyed is king

d_6. A friend in need is a friend indeed



Term	F_t	*postings*
better	1	(d_3,1)
bird	3	(d_1,1); (d_2,1);(d_4,1)
blind	2	(d_3,1) ;(d_5,1)
bush	1	(d_1,1)
catch	1	(d_4,1)
early	1	(d_4,1)
eye	3	(d_3,1);(d_5,1);(d_7,2)
feather	1	(d_2,1)
flock	1	(d_2,1)
friend	1	(d_6,2)
hand	1	(d_1,1)
indeed	1	(d_6,1)
king	1	(d_5,2)
need	1	(d_6,1)
one	2	(d_3,1);(d_5,1)
quite	1	(d_3,1)
together	1	(d_2,1)
tooth	1	(d_7,2)
two	1	(d_1,1)
worm	1	(d_4,1)
worth	1	(d_1,1)

Figure 5: Real Inverted Index for the "proverbs" data set

2.2.2.2 Building the inverted index

Different methods have been proposed to construct the inverted index on the typical available hardware (i.e., hard disk storage and at least a few megabytes of RAM memory):

- ***In-memory inversion***

This method [15] [16] consists in making a first pass over the collection, computing for each term its frequency in the collection, allocating the corresponding space necessary for the index construction in the RAM memory, and doing a second pass over the collection to construct the inverted index. This technique is very fast to implement, the query time is very fast also, and the in-memory index construction avoids any index fragmentations. But the main problem with this technique is that it is not scalable with the increasing size of the collection given the limited size of the RAM memory.

- ***Disk-based inversion:***

To deal with limited RAM constraints and the increasing size of the data collection, techniques based on disk storage use have been developed:

- ***Sort-based inversion***

To solve the problem of indexing a large collection which does not fit in main memory, the *sort-based inversion* technique [17] [18] creates the index vocabulary in the RAM memory and allocates a temporary file on the disk to record triples of form (t, d, $f_{d,t}$), where t is the term in the document d with the frequency $f_{d,t}$. Each document in the collection is converted to triples and stored in the temporary file on the disk. Then, this temporary file is sorted progressively by chunks that fit into the RAM memory, firstly in the descending order of the terms and secondly on descending order of the document identifiers. Finally, this sorted temporary file is used to generate the final inverted index.

- ***Merge-based inversion:***

The increasing size of the document collection can prevent storing the entire vocabulary in memory. To deal with this problem, the index is split in several partitions, each partition having the same internal structure as an inverted index. Once the partition in the RAM memory reaches the RAM limit, it is stored on the disk (including its vocabulary) and a new partition is created for the following documents of the collection. The partitions on the disk are merged hierarchically to reach one inverted index structure. The merge-based inversion [15] [19] is the ideal method for any RAM and collection size, requiring a single parsing of each document in the collection.

However, in the context of secure tokens, the tiny RAM constraint makes all the above methods inefficient. Therefore, we propose a specific method to build and maintain the index structure in stream. Our approach shares a few similarities with the merge-based inversion, i.e., build small inverted indexes in RAM and periodically merge them with the inverted indexes in the Flash storage (see Chapters III and IV). But in our case, the indexes in RAM and in Flash have specific structures to permit an efficient merge with low RAM usage. Also, the partitions stored in Flash have a specific organization to obtain a good tradeoff between the merge cost and the query cost.

2.2.2.3 Inverted index maintenance

Adding, removing or updating a document to/from/in the collection requires to modify the inverted index accordingly as well. Several techniques for updating the inverted index have been proposed:

- ***Rebuild***

The simplest way to deal with updates is to reconstruct the inverted index from scratch [20]. For some applications that need to be available all the time like a web site for example, rebuilding the inverted index from scratch is more convenient than modifying the index, which can make the index temporarily unavailable. Also, when the updates are important, updating the index structure is more costly and time consuming than rebuilding the index from scratch. Yet, rebuilding the index from scratch is less suitable if the updates are frequent and the data collection is huge. When new documents need to be added or the number of updates reaches a predefined value, a new index is constructed to include the new documents. The old index on the disk is deleted and replaced with the new generated index.

- ***Intermittent merge***

With this technique the updates are buffered and indexed in memory to accelerate search queries [21]. When the updates reach a RAM limit, they are merged with the inverted index. During the merge the index can still be used by queries at the cost of temporarily having two index copies on disk. The new updates are either blocked or temporarily indexed in the RAM memory. With this approach, the updates are fast and the queries give correct answers during the inverted index lifetime. In addition, this technique is suitable for any size of data collection and RAM memory.

- ***Incremental update***

Another alternative is to update the inverted index in stream, i.e., for every insert/delete/update operation the inverted index is updated to include the changes term by term and list by list using random writes. The in-place index updating [22] can be less efficient than the intermittent merge because of the large cost and number of random writes. Alternative solutions propose to reduce the number of random writes by buffering the updates before committing them in batch. But this solution may still be less efficient than the intermittent merge where the data is written sequentially.

For embedded systems, rebuilding the index from scratch for each update is very costly in terms of the number IOs especially when taking into account the limited size of the RAM memory. On the other hand, modifying the index in-place at each document update generates many random writes in the NAND Flash memory resulting in very large cost of the updates. Indeed, since a document can contain an important number of terms (e.g., hundreds to thousands), a document insertion/deletion/update requires to modify the inverted list associated to each document term.

2.3 Search

The main focus of IR systems is to extract a subset of relevant documents from the data collection, to answer users' queries according to their information needs. Several models have been proposed. The main idea in these models is to compute a score for the documents using a scoring function based on the user query and the content of the documents.

2.3.1 Baseline Query Evaluation

The core problem is, given a collection of documents and a user query expressed as a set of terms $\{t_i\}$, to retrieve the k most relevant documents according to a ranking function. In the wide majority of the related works, the *tf-idf* score [23], i.e., term frequency-inverse document frequency, is used to rank the query results. For a query $Q=\{t\}$, the *tf-idf* score of each indexed document d containing at least a query term can be computed as follows:

$$tf-idf(d)=\sum_{t\in Q}\log\left(f_{d,t}+1\right)\cdot\log\left(\frac{N}{F_t}\right)$$

where $f_{d,t}$ is the frequency of term t in document d, N is the total number of indexed documents, and F_t is the number of documents that contain t in the document collection. This formula is given for illustrative purpose, the weight between $f_{d,t}$ and N/F_t varying depending on the proposals [24].

Evaluating the query $Q=\{t\}$ is described in Figure 6: (i) by accessing the I.S index search structure (see Figure 4) to retrieve for each query term t the inverted list element $\{I.L_t\}t\in Q$; (ii) allocating in RAM one container A_d for each unique document identifier d in these lists; (iii) computing the score of each of these documents using a weight function, e.g., *tf-idf*; (iv) ranking the documents according to their score and producing the k documents with the highest scores.

However, in an embedded context given the limited RAM constraint, allocating one container in the RAM memory for each document identifier mentioned in t's inverted list is not possible if the document collection is large. Therefore, the typical query evaluation has to be reconsidered for the embedded context (see Chapters III and IV).

Top-k Algorithm

Let N, be the total number of documents in the collection,

Let Q be the user query, $Q=\{t_1,t_2,t_3,t_4,\ldots t_n\}$

1. Allocate in RAM one container A_d for each document identifier *"d"* in $I.L_t$ list, set $A_d \leftarrow 0$.
2. **foreach** term *t* in *Q* **do**
3. Compute the IDF value $IDF = log\ (N/F_t)$;
4. Fetch *t* inverted list $I.L_t$,
5. **foreach** posting $(d, f_{d,t})$ in $I.L_t$ **do**
6. Compute the TF-IDF score ; $S_t = log(1 + f_{d,t})*log\ (N/F_t)$
7. Add , $A_d \leftarrow A_d + S_t$
8. Rank the A_d values and produce the *k* documents with the highest score.

Figure 6: Top-k query evaluation algorithm

2.3.2 Term Weighting

Term weighting consists in giving each term in a document a weight to reflect its importance to the document and the collection. Generally, all the techniques developed for term weighting compute two statistical values for each term: a local term weighting to represent importance of the term inside the document, and a global term weighting to measure the overall representativeness of the term in the collection.

- ***Local term weighting*** **($w_{d,t}$)**:

It reflects the representativeness of the term inside the document. The most known local term weighting functions are [24]:

- *Term frequency* (TF): it measures the occurrence frequency of the term *t* in the document *d*. Since documents can have variable lengths, this measure can be normalized by dividing it to the length of the document (the total number of terms in the document).
- *Boolean function*: the term weight is equal to $w_{d,t}=1$ if the term *t* appears at least once in the document *d*, and to $w_{d,t}=0$ otherwise.
- *Logarithmic function*: to reduce the difference of terms' weights between documents, the term frequency is combined with a logarithm function, $w_{d,t} = Log(f_{d,t}) + \alpha$, where is a constant.

- ***Global term weighting*** ($w_{q,t}$):

It measures the importance of the term over the collection. The idea is to give a larger term weight to the terms that appear less frequently in the collection, and give less weight to the terms that are frequent among the documents in the collection. This weight is also called *inverse document frequency* (*IDF*) and it is computing as follow:

$$W_{q,t} = \log \frac{N}{F_t}$$

Where F_t is the number of documents containing the term *t* in the collection, and *N* is the total number of the documents in the collection.

The double term weighting, local and global, are combined to produce the *TFxIDF* term weighting, which increases proportionally with the number of times the term appears in a document and it decreases with the number of appearances of the term in the collection. Hence, the score of the query $q=\{t\}$ and the documents *d* is computed as:

$$S_{q,d} = \sum_{t} w_{q,t} * w_{d,t}$$

2.4 Optimizations

2.4.1 Storage Optimization

Several techniques have been proposed to optimize the inverted index size on disk in order to reduce the disk traffic and disk space consumption, and thus accelerate the query search.

Inverted lists compression. An efficient representation of postings (document identifier and term frequency) can be made by coding these integer values in a compact format rather than a fixed-width representation (e.g., 32-bit or 16-bit). Integer codding like unary [25], Elias' gamma [26], and Golomb [27], can be used to represent the document identifier and the term frequency. However, the inverted lists' compression comes with the disadvantage related to decoding the posting lists before any use. Also, if the postings are updated, they need to be recoded. Therefore, the inverted lists' compression is not suited for embedded systems which have low CPU power, since in this case the decompression time can be important.

2.4.2 Memory Optimization

To reduce memory consumption and query execution time, several techniques were suggested to satisfy this purpose.

- ***Reducing memory consumption***

Accumulator limiting. Limiting the total number of document accumulators in the memory by allocating accumulators only to documents with high relevance for the query can save greatly the space memory [28] [29]. Common words like ("the", "a", "an", etc.) are of low importance in a search query, and could be ignored. Also, by limiting allocated memory accumulators only to documents with low term frequency F_t, the memory consumption and the search performance and result relevance can be greatly improved.

Accumulator thresholding. Another approach for limiting the number of document accumulators in the memory is by creating document accumulators only for documents with a TFxIDF score exceeding a certain threshold S [30] [31]. The value of the threshold S can be dynamically increased during the query evaluation (e.g., after a significant number of documents having high scores have been retrieved), and thus reduce even more the memory consumption. During the query evaluation, the accumulators of documents with a smaller score than the threshold S are discarded.

Considered in the embedded context, such optimizations are largely insufficient and cannot solve alone the problem of the tiny RAM of the MCU.

- ***Reducing retrieval costs***

Frequency-ordered inverted lists. The frequency-ordered inverted lists approach stores the inverted lists sorted in decreasing order of the term frequency $f_{d,t}$ values of the postings [32] [33]. The idea is that the query evaluation will consider first the documents with large $f_{d,t}$ values, and therefore, having potentially the largest scores. An inverted list is processed until the $f_{d,t}$ values become smaller than some predefined threshold S. Then, the query process discards the remaining part of the inverted list to accelerate the evaluation.

This approach reduces the query cost but has the opposite effect on the insertion/update cost, since the inverted lists have to be maintained sorted on the $f_{d,t}$ values of the postings.

2.5 Index Efficiency Measurements

The search engine efficiency is typically assessed using five measures:

1. **Indexing time:** the time needed to insert the data collection.

2. **Indexing space:** the disk space needed to construct the index during the insertion of the data collection.
3. **Indexing storage:** the disk space needed to store the index, after the insertion of the data collection.
4. **Query/update latency:** the time needed to answer a query or to process an update.
5. **Index throughput:** number of index operations (i.e., queries and updates) the search engine is able to process in one second.

2.6 Access Control for Documents

Traditional access control models are the Mandatory Access Control (MAC), where labels are associated to objects/subjects, the Discretionary Access Control (DAC), a decentralized user-based mechanism where the creator of a data defines the set of authorizations, and the Role-Based Access Control (RBAC), where roles are assigned to users and permissions are assigned to those roles [34]. RBAC has then been extended with temporal or geographical constraints or additional concepts to structure the policies (e.g. team, task, organization). Usage CONtrol model (UCON) express finer control on data usage, a prerequisite when privacy (and not only confidentiality) needs to be protected. In this model, a usage control decision is determined by combining authorizations, obligations, and conditions [35]. In this spirit, [36] has proposed a purpose-based AC model. Finally, Attribute-based access control (ABAC) allows the definition of policies which combine attributes together (user attributes, resource attributes, environment attribute etc.) [37].

Tag-based access control (TBAC) combines the ease of use of extensional systems like (MAC, DAC, RBAC, etc.) with the semantic policy specification, while still maintaining a meaningful degree of the expressiveness of logical systems [38] [39]. Tag-based access control systems use tags to express policies. Tags are words or phrases assigned to pieces of content, added by the content owner or extracted dynamically. Tags can be applied only on subjects (e.g., users and applications) or objects (e.g., documents, photos, posts, videos) or on both. In [40] [41], the authors propose an access control for web context based on TBAC. The authors motivate the use of TBAC for web applications by: (i) the flexibility of the model to use with dynamic data; (ii) the varying levels of technical knowledge for web users; (iii) the fact that web

users are familiar with using tags; (iv) tags are supported by a very large number of applications in the web; and (v) TBAC policies are easy to transfer across sites.
A TBAC model for blogs by reusing tags found in blog posts to construct access control policies is proposed in [40] . The authors demonstrate by a user study that TBAC can provide accurate, flexible, and usable policies by using only a few tags per policy. In [41], the authors provide a semantic access control for online photo albums based on descriptive tags found in social networks such as Flickr, Delicious and linked data. In [39] [42] [43] [44], Mazurek et al. propose a secure access control model to share files based on TBAC models and carry out several users studies to measure the usability of such models by non-specialist users. For instance, the work in [44] explains the needs of home users (i.e., a non-technical computer users) to an access control that suits them. Home users are technically inexperienced and impatient with complex access control. Also, there is no system administrator in home context to maintain access control policies, and users require dynamic access control policies that can be changed quickly as documents move through their life cycle. In [39], a user study is performed that shows that tags related to data organization of home users can be reused to express coherent policies for access control. In addition, the work in [39] provides a secure access control model for sharing files in a distributed system. In [43], a user study is carried out to better understand the future needs of a reactive access control to help non-technical home users to easily manage their access control. Recently, a few works have proposed tag-based access control (TBAC) models in the context of personal data management and sharing. For example, applying TBAC for sharing semantic data in PIMs (Personal Information Manager) was outlined in [45].

Tags can also be applied on subjects. For instance, in [46] the authors perform a study in which users are trusted with adding tags to each other to share resources in collaborative work environment. Finally, tags can be applied on both subjects and objects. The work in [38] demonstrates that most traditional access control models (e.g., RBAC, MAC, DAC) can be emulated by a TBAC model and give an example of a high security level domain by applying TBAC in a military context.

A non-technical user is the common case in the personal cloud context. Therefore, we apply a TBAC model to the proposed search engine in order to give the users the flexibility and easiness to administrate the file sharing across multiple applications and users.

3. Embedded Search Engine

A few pioneer works recently demonstrate the interest of embedding search engine techniques into smart objects equipped with extended Flash storage to manage collections of files stored locally [2] [6] [7] [11]. These works rely on a similar design of the embedded inverted index as explained in Section 3.1. Traditional IR systems were designed for powerful devices, i.e., having at least MBs of RAM memory and using magnetic disk storage. Many studies like [47] [48] address the problem of storage and indexing in NAND flash, but important gains in terms of flash write cost require large buffers or logs in RAM. For an MCU linked to a large personal database in NAND flash, the RAM consumption and random write cost are always conflicting. In this section, we present a few recent works like Microseach [2] [6] that deal with the problem of embedding a search engine in smart devices. The idea in these works is to resolve this intrinsic conflict between tiny RAM and NAND flash storage by organizing the index into a purely sequential log index in flash.

3.1 Microsearch

Microsearch [2] [6] is a full-text search engine for embedded devices used in pervasive computing. Microsearch is designed for small embedded systems that can be found in the physical world, such as sensors, mottes, PDAs, which are equipped with a tiny RAM of a few KBs and Flash storage. Microsearch adapts IR techniques to the embedded smart objects constraints to index and query data, and answer users queries with top-k relevant responses.

3.1.1 Design

Microsearch is composed of two memory structures (i.e., a *buffer cache* to receive metadata from inserted files and an *inverted index* to query the recently inserted data), and a *log-structure* in flash to persistently store the indexed metadata (see Figure 7).

3.1.1.1 Insertion

At the insertion of a new file, Microsearch stores it in the flash memory and reorganizes its metadata to triples of < t, $f_{d,t}$, d_adr >, where t is a term in the file d, $f_{d,t}$ is the weight of the term in the file d and d_adr is the address of the file d in the flash memory. The triples are temporarily stored in the *buffer cache* memory. This process continues as more files arrive. When the *buffer cache* is full, it is flushed in the *log-structure* and the *hash table* updated with the inverted lists addresses.

Buffer cache: it is the largest structure in the RAM memory and contains all the inserted file metadata triplets (ex.: $<t_1, f_{d1,t}, d\_adr_1>,<t_1, f_{d2,t}, d\_adr_2>….<t_n, f_{d5,t}, d\_adr_5>,<t_n, f_{d10,t}, d\_adr_{10}>,<t_n, f_{dn,t}, d\_adr_{11}>.$). Inside the *buffer cache* the triplets are grouped by term.

Buffer cache eviction (the log-structure): when the buffer cache is full, it is flushed in flash to free space for the next file insertions. Instead of maintaining one inverted list per term in the dictionary (like in the classical inverted index), each term is hashed to a bucket and a single inverted list is built for each bucket. The inverted lists are stored sequentially in the Flash memory, within chained pages, and only a small hash table referencing the last Flash page of each inverted list is kept in RAM. This approach complies with a small RAM and suits well the Flash constraints by precluding fine grain random (re)writes in Flash.

Hash table: it is a small hash table in the RAM memory referencing the address of the last Flash page of each bucket. The number of buckets is kept small, such that (i) the large dictionary of terms (usually tens of MB) is replaced by a small hash table stored in RAM, and (ii) the main part of the RAM can be used as an insertion buffer for the inverted lists elements, i.e., $<t, f_{d,t}, d\_adr>$ triple. However, the small number of buckets has a negative impact on the query performance since each inverted list corresponds to a large number of different terms. Therefore, the query process retrieves more index data than the relevant information to compute the query result.

Data deletion: Microsearch does not support document deletions, but only data aging mechanisms, where old index entries automatically expire when overwritten by new ones.

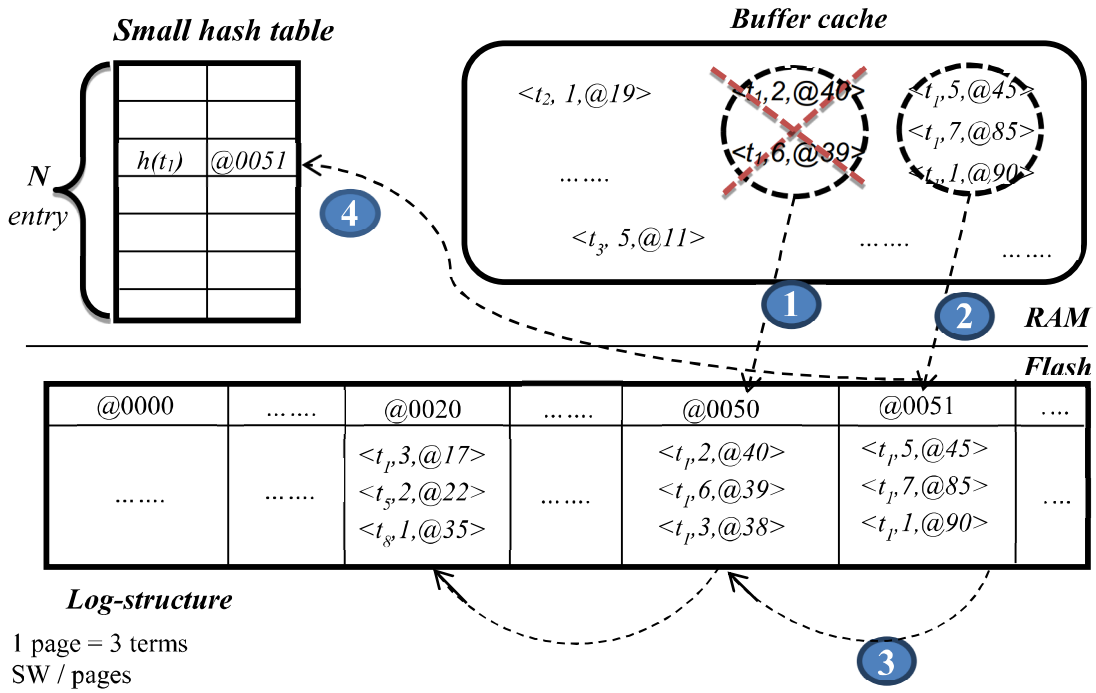


Figure 7: Microsearch index structure

3.1.1.2 Query

To evaluate a query $Q=\{t\}$, Microsearch uses the typical TFxIDF scoring function to return the top-k relevant results. The query evaluation is performed in two steps (see Figure 8):

- Compute the IDF value for each query term.
- Compute the TFxIDF score in a second a step.

The query evaluation starts by searching the buffer cache for each query term triples. If such triples are found, Microsearch keeps them in a separate place in the RAM memory. Then, Microsearch hashes each query term to obtain its address of the last bucket in the flash memory. A first step is required to calculate the IDF for each query term, by accessing each inverted list and computing the number of documents corresponding to the query term (other terms found in the chained index pages not corresponding to the queried term are discarded). In a second step, Microsearch computes the TFxIDF for each document, by loading in RAM one index flash page for each query term. To compute the TFxIDF score of a document, all the query terms in the document must be loaded in RAM at the same time. To solve this challenge, Microsearch uses the following insertion strategy: "if a document d_2 is inserted after a document d_1, the storage address of the document d_2 is greater than the document d_1"

following the sequential writing process in the flash memory. For each term page loaded in the RAM memory, Microsearch calculates the minimum address of the documents in the triples, and then calculates a threshold *th* between the pages equal to the maximum address of the minimum address of the documents in each page (i.e., $th = max\ [min_{adr\_doc}\ (t_i)]$, $\forall\ t \in Q$). This threshold defines a lower limit to documents having all their queried terms in the RAM memory at the same time. Since all the documents having a storage address bigger than *th* have all their queried terms in the RAM memory, Microsearch proceeds with the score computation of these documents using the TFxIDF formule, and keeps in memory the *k* best scored document ids. When a term page in RAM is completely read, the next page in the inverted list is loaded in RAM and a new *th* is computed. This is repeated until the end of all inverted lists is reached. Finally, Microsearch returns the best top-k scores and their corresponding document ids.

Note that, since each inverted lists corresponds to a large number of different terms, this unavoidably leads to a high query evaluation cost that increases proportionally with the size of the data collection. Typically, the less RAM is available for the hash table in RAM, the smaller the number of hash buckets and the more severe the problem is.

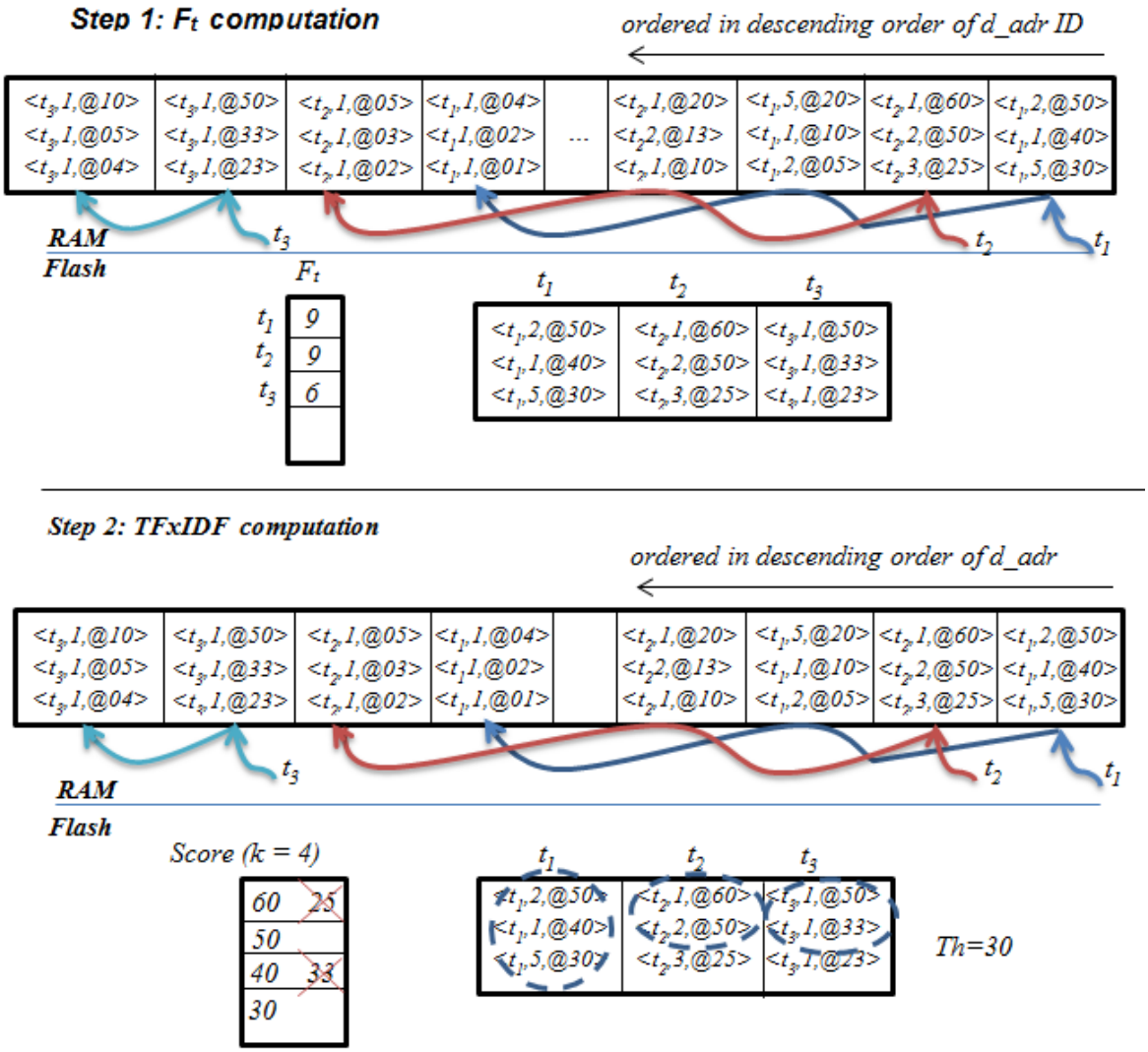


Figure 8: Query evaluation in Microsearch

3.1.2 Microsearch Performance

To evaluate the insertion and query performance, Microsearch uses the telosB mote [49] as a representative embedded system hardware architecture with 10KB of RAM and 1MB of Flash memory. In their experiments, the RAM consumption of the search engine is fixed to 3KB, shared between the hash table and the buffer cache, with a workload of one thousand documents.

To measure the insertion and query performance, the number of buckets is varied from 2 to 32 buckets. The insertion time is faster with a smaller number of buckets. Generally, the insertion time varies from 4s with a hash table of 2 buckets to 11s with a hash table of 32 buckets to insert the entire dataset of one thousand documents. The

query time has an opposite behavior, i.e., the query time is faster with a larger hash table. The measured query times are less than 2 seconds for the whole dataset no matter the index configuration.

3.2 Other Works Similar to Microsearch

A similar design is proposed in [11] to build a distributed search engine to retrieve images captured by camera sensors. A local inverted index is embedded in each sensor node to retrieve the relevant images locally, before conducting the distributed search. However, this work considers powerful sensors nodes (with tens of MB of local RAM) equipped with custom SD card boards (with specific performances). At the same time, the underlying index structure is based on inverted lists organized in a similar way as in [2].

Snoogle [7] is a distributed search engine for the physical world that helps people to search for physical objects in their surroundings. Snoogle employs Microsearch to index data inside a smart object (e.g., a sensor). Then, Snoogle calculates distributed top-k queries on top of wireless sensor networks by using a hierarchical organization of the network (i.e., some network nodes are super-peers that centralize the query results of the nodes in their vicinity and re-transmit the partial results to other supr-peers). Also, Snoogle uses Bloom filters to reduce the data transmission over the network.

Other query evaluation facilities have been recently proposed for smart objects equipped with large Flash memory [50] [8] to enable filtering operations. For instance, relational database operations like selection, projection and join have been proposed for sensors in [51] and for new generations of SIM cards with large Flash storage capacities in [5].

3.3 Conclusion

All these methods are highly efficient for document insertions, but fail to provide scalable query processing for large collections of documents. Indeed, a small number of buckets in the inverted index provides a fast insertion time with long inverted lists, while it increases significantly the query time. The inverted lists are not selective with respect to queries and each queried inverted list needs to be read twice to calculate first the IDF and then the TFxIDF score. Therefore, the usage of these methods is limited to applications that require storing only a small number (few hundreds) of

documents. In addition, these techniques do not deal with documents deletions and updates, but consider only document expiration.

4. Other Related Indexing Techniques

4.1 B-tree Indexing in NAND Flash

In the database context, adapting the B-tree to NAND Flash has received a great attention. Indeed, the B-tree is a very popular index and its standard implementation performs poorly in Flash [52]. Many recent proposals [47] [48] [52] tackle this problem or the closely related problem of a key-value stored in Flash [53].

The key idea in these approaches is to buffer the updates in log structures that are written sequentially and to leverage the fast (random) read performance of Flash memory to compensate the loss of optimality of the lookups. When the log is large enough, the updates are committed into the B-tree in a batch mode, to amortize the Flash write cost. The log must be indexed in RAM to ensure performance. The different proposals vary in the way the log and the in-memory index are managed, and in the impact it has on the commit frequency. Here below, we give a couple of prominent examples.

The FD-Tree [48], a *logarithmic and fractional cascading* structure, is designed to optimize the traditional B-tree for SSDs (solid state disk drives). FD-tree consists of an in-memory head tree (i.e., a small B-tree) and of a few *cascading* sorted runs flash of exponentially increasing size stored in flash. The insertions and the updates are firstly integrated in the head tree. When the head tree attains a predefined limit in RAM, it is merged with the first sorted run in the flash storage. Similarly, if a sorted run reaches the predefined limit, it is merged with the subsequent, larger sorted run on the SSD. Therefore, the objective of the in-memory head tree of the FD-tree is to transform the costly random writes into sequential writes in flash. However, to have good performance, the FD-tree [48] requires about 512KB to 8 MB of RAM memory for the head tree. But this RAM consumption is too large for embedded systems and makes this structure inappropriate to our context.

The LA-Tree (Lazy-Adaptive Tree) [47] is another B-tree-like structure optimized for flash storage. The idea used by the LA-tree is to amortize the update cost by using cascaded buffers. To this end, LA-tree uses flash-resident buffers, which are attached to various levels of the B+-tree to log the updates. The updates are integrated in the

B+-tree only when the leaf level buffers become full. The benefit of lazy updates is that they reduce the total number of flash accesses since they are done in batch. To accelerate the look-up operations, an online algorithm is proposed to resize the buffers in the RAM memory according to the workload. Larger buffers are used for update-intensive workloads, while smaller buffers are more efficient in the case of lookup-intensive workloads. However, as in the case of the FD-tree, the LA-tree requires too much memory (with respect to the available RAM of smart objects) to achieve good update scalability.

In conclusion, to amortize the write cost by a significant factor, the log must be seldom committed, which requires more RAM. Conversely, limiting the RAM size leads to increasing the commit frequency, thus generating more random writes. The RAM consumption and the random write cost are thus conflicting parameters. Under severe RAM limitations, the gain on random writes definitely vanishes.

4.2 Partitioned Indexes

In another line of work, partitioned indexes have been extensively employed especially to improve the storage performance in environments with insert-intensive workloads and concurrent queries on magnetic disks. A prominent example is the LSM-tree (i.e., the Log-Structured Merge-tree) [54] and its many variants (e.g., the Stepped Merge Method [55], the Y-tree [56], the Partitioned Exponential file [57] , and the bLSM-tree [58] to name but a few).

The LSM-tree consists in one in-memory B-tree component to buffer the updates and one on-disk B+-tree component that indexes the disk resident data. Periodically, the two components are merged to integrate the in-memory data and free the memory. The benefit of such an approach is twofold. First the updates are integrated in batch, which amortizes the write cost per update. Second, the merge operation uses sequential I/Os, which reduces the disk arm movements and thus, highly increases the throughput. If the indexed dataset becomes too large, the index disk component can be divided into several disk components of exponentially increasing size to reduce the write amplification of merges. Many works have proposed optimized versions of the LSM-tree. For instance, bLSM [58] fixes several limitations of the LSM-tree. Among the improvements, the main contribution is an advanced merge scheduler that bounds the index write latency without impacting its throughput. Also, the FD-tree [48] (see the previous section) proposes a similar structure with the LSM-tree to optimize the data indexing on SSDs.

Furthermore, the storage systems of the major web service providers, e.g., Google's Bigtable and Facebook's Cassandra, employ a similar partitioning approach to implement key-value stores. The idea is to buffer large amounts of updates in RAM and then flush them in block on disk as a new partition. Periodically, the small partitions are merged into a large partition.

The search engine proposed in this thesis shares the general idea of index partitioning. However, the similarity stops at the general level because of major differences regarding the targeted hardware platforms (embedded systems versus high-end servers) and the queries of interest (top-k keyword search versus classical key-value search). Consequently, our solution differs from the above mentioned ones in a number of aspects (e.g., RAM usage, partitioning organization, management of updates and deletions, way of computing the top-k with a minimal RAM consumption).

5. Conclusion

In this chapter, we analyzed first the hardware characteristics of secure tokens, which oppose a scarce RAM memory against large NAND Flash storage that badly supports fine-grained random writes. Then, we analyzed the requirements of a full-text search engine for document search, and explained the internal structure of the inverted index and the typical techniques to process insertions/updates and searches. In a third part, we presented the state of the art related to the tag-based access control models for document sharing. Finally, we discussed the state of the art concerning embedded search engines, and presented in detail Microsearch, which is the representative full-text search index method for embedded devices.

With the constraints of secure tokens in mind, we then surveyed the state of the art related to indexing and storage techniques for Flash memory, and found that none of them could meet all the requirements of secure tokens. Thus, the challenge lies in the combination of a tiny RAM memory with a huge NAND Flash persistent storage that badly accommodates fine-grained random writes.

To execute queries with a tiny RAM, we need to massively index the dataset to allow evaluating the queries in pipeline (i.e., with a bounded amount of RAM that does not depend on the dataset size). On the other hand, massively indexing the data leads to a proportional increase in the number of random writes to maintain the index structure, which has inacceptable cost in NAND flash memory. To alleviate the large update cost, the typical approach is to buffer the updates in RAM and commit them in batch to amortize the cost of an update. But buffering the updates is not possible in the

embedded context. Therefore, tiny RAM and large NAND flash storage lead to conflicting constraints (see Figure 9) that cannot be reconciled by the existing indexing methods to achieve both update and query scalability at the same time. In the next chapter, we classify the state of the art techniques for document indexing under these hardware constraints, formulate the problem and define our proposed design principles to solve this problem.

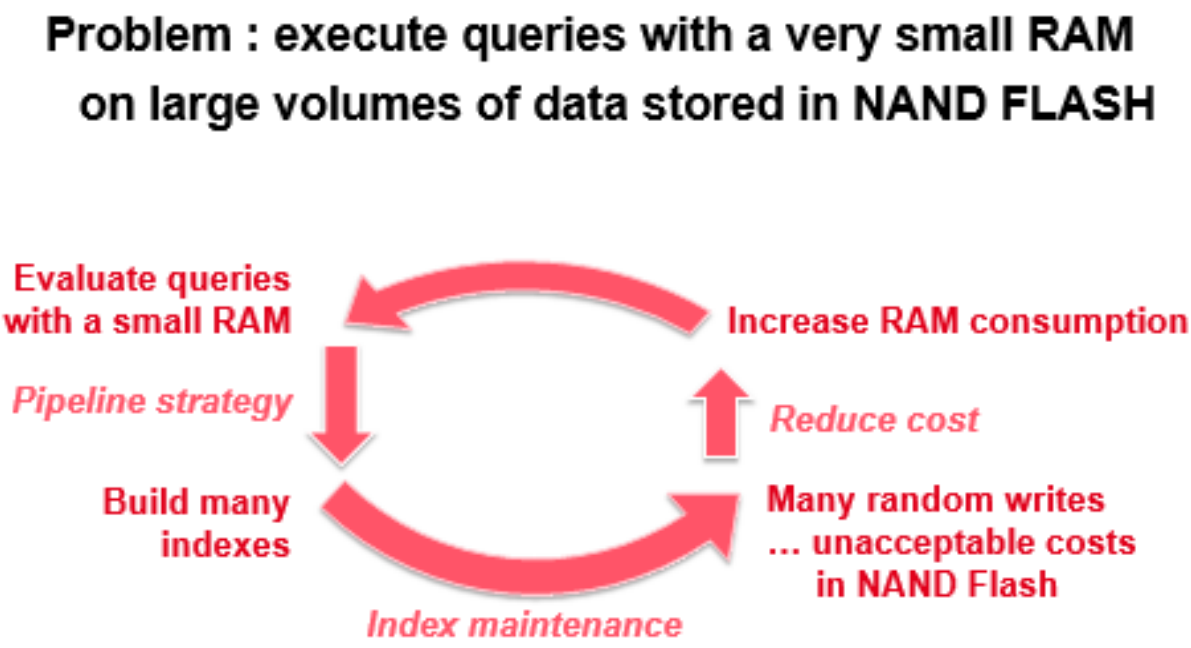


Figure 9: Tiny RAM and large NAND flash storage lead to conflicting constraint

Chapter III
Problem Formulation and Design Principles

In this chapter, we start by classifying the state of the art indexing methods with respect to the hardware constraints of smart objects and subsequently formulate our problem. We then introduce our design principles (i.e., Write-Once Partitioning, Linear Pipelining and Background Linear Merging) to define a search engine that is RAM Bound and reaches both insertion and query scalability. Finally, we define the access control integration with the search engine for document sharing.

1. Indexing Methods and Smart Objects

Tiny RAM and NAND Flash persistent storage introduce conflicting constraints which lead to split state of the art solutions in two families. On the one hand, the *insert-optimized family* reaches insertion scalability thanks to a small indexed structure buffered in RAM and sequentially flushed in Flash, thereby precluding costly random writes in Flash. This good insertion behavior is however obtained to the detriment of query scalability, the performance of searches being roughly linear with the index size in Flash. This family includes the methods presented in Section 3 (Chapter II), e.g., Microsearch [2], Snoogle [7] or CameraSensors [11].

Conversely, the *query-optimized family* reaches query scalability by adapting traditional indexing structures to Flash storage, to the detriment of insertion scalability, the number of random (re)writes in Flash (linked to the log commit frequency) being roughly inversely proportional to the RAM capacity. This family includes the classical inverted index [24] (see Section 2.2) and the methods presented in Section 4 (Chapter II). In addition, we are not aware of works addressing the crucial problem of random document deletions in the context of an embedded search engine. Figure 10 illustrates the two families and their approximate performance with queries and updates.

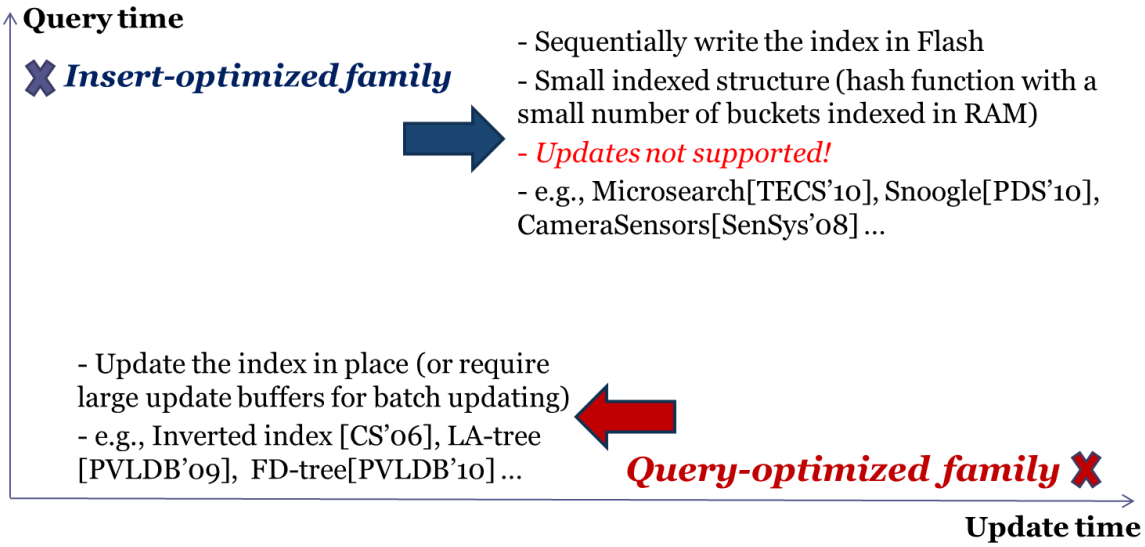


Figure 10: Indexing methods families in the context of embedded systems

2. Problem Formulation

In the light of the preceding sections, the problem addressed in this thesis can be formulated as *designing an embedded full-text search engine* that has the following three properties:

- *Bounded RAM agreement:* the proposed engine must be able to respect a predefined RAM consumption bound (RAM_Bound), precluding any solution where this consumption depends on the size of the document set.
- *Full scalability:* the proposed engine must be scalable for queries and updates (insertion, deletion of documents) without distinction.
- Security: the proposed engine must filter out all documents violating the access control (AC) policy defined by the user. Moreover, the AC evaluation should not hamper the previous two properties namely, RAM Bound and insertion/query scalability.

The Bounded RAM agreement is required to comply with the widest population of smart objects. The consequence is that the full-text search engine must remain functional even when very little RAM (a few KB) is made available to it. Note that the RAM_Bound size is a subpart of the total physical RAM capacity of a smart object considering that the RAM resource is shared by all software components running in parallel on the platform, including the operating system. The RAM_Bound property is also mandatory

in a co-design perspective where the hardware resources of a given platform must be precisely calibrated to match the requirements of a particular application domain.

The Full scalability property guarantees the generality of the approach. By avoiding to privilege a particular workload, the index can comply with most applications and data sets. To achieve update scalability, the index maintenance needs to be processed without generating random writes, which are badly supported by the Flash memory. At the same time, achieving query scalability means obtaining query execution costs in the same order of magnitude with the ideal query costs provided by a classical inverted index (see Figure 4).

The Security of the whole approach lies in the fact that both the index and the access control engine are embedded in the secure token as explained in Chapter II. The reason is that the secure MCU offers high security guarantees for the executed code based on the difficulty of tampering or observing the data processing. Hence, to preserve these good safety properties, we need to evaluate the AC rules inside the secure hardware as well. Delegating the AC control evaluation outside the secure hardware leads to accept externalizing sensitive data outside the secure area, which decreases the overall security of the search engine.

2.1 Design Principles

Satisfying the Bounded RAM agreement and Full scalability properties simultaneously is challenging, considering the conflicting MCU and Flash constraints mentioned above. To tackle this challenge, we propose in this thesis an indexing method that relies on the following three design principles.

P1. Write-Once Partitioning: *Split the inverted index structure I in successive partitions such that a partition is flushed only once in Flash and is never updated.*

By precluding random writes in Flash, Write-Once Partitioning aims at satisfying update scalability. Considering the Bounded RAM agreement, the consequence of this principle is to parse documents and maintain *I* in a streaming way. Conceptually, each partition can be seen as the result of indexing a window of the document input flow, the size of which is limited by the RAM_Bound. Therefore, *I* is split in an infinite sequence of partitions $<I_0, I_1, \ldots, I_p>$, each partition I_i having the same internal structure

as I. When the size of the current I_i partition stored in RAM reaches RAM_Bound, I_i is flushed in Flash and a new partition I_{i+1} is initialized in RAM for the next window.

A second consequence of this design principle is that document deletions have to be processed similar to document insertions since the partitions cannot be modified once they are written. This means adding compensating information in each partition that will be considered by the query process to produce correct results.

P2. Linear Pipelining: *Compute each query Q with respect to the Bounded RAM agreement in such a way that the execution cost of Q over* $<I_0, I_1, \ldots, I_p>$ *is in the same order of magnitude as the execution cost of Q over I.*

Linear Pipelining aims at satisfying query scalability under the Bounded RAM agreement. A unique structure I as the one pictured in Figure 4 is assumed to satisfy query scalability by nature and is considered hereafter as providing a lower bound in terms of query execution time. Hence, the objective of Linear pipelining is to keep the performance gap between Q over $<I_0, I_1, \ldots, I_p>$ and Q over I, both small and predictable (bounded by a given tuning parameter). Computing Q as a set-oriented composition of a set of Q_i over I_i, (with $i=0,...p$) would unavoidably violate the Bounded RAM agreement as p increases, since it will require to store all Q_i's intermediate results in RAM. Hence the necessity to organize the processing in pipeline such that the RAM consumption remains independent of p, and therefore of the number of indexed documents. Also, the term linear pipelining conveys the idea that the query processing must preclude any iteration (i.e., repeated accesses) over the same data structure to reach the expected level of performance. This disqualifies brute-force pipeline solutions where the tf-idf scores of documents are computed one after the other, at the price of reading the same inverted lists as many times as the number of documents they contain.

However, Linear Pipelining alone cannot prevent the performance gap between Q over $<I_0, I_1, \ldots, I_p>$ and Q over I to increase with the increase of p as (i) multiple searches in several small $I_i.S$ are more costly than a single search in a large $I.S$ and (ii) the inverted lists in $<I_0, I_1, \ldots, I_p>$ are likely to occupy only fractions of Flash pages, multiplying the number of Flash I/Os to access the same amount of data. A third design principle is then required.

P3. Background Linear Merging: *To limit the total number of partitions, periodically merge partitions in a way compliant with the Bounded RAM agreement and without hurting update scalability.*

The objective of partition merging is therefore to obtain a lower number of larger partitions to avoid the drawbacks mentioned above. Partition merging must meet three requirements. First the merge must be performed in pipeline to comply with the Bounded RAM agreement. Second, since its cost can be significant (i.e., proportional to the total size of the merged partitions), the merge must be processed in background to avoid locking the index structure for unbounded periods of time. Since multi-threading is not supported by the targeted platforms, background processing can simply be understood as the capacity to interrupt and recover the merging process at any time. Third, update scalability requires that the total cost of a merge run be always smaller than the time to fill out the next bunch of partitions to be merged.

2.2 On-the-fly Access Control Monitoring

An owner of a personal cloud such as Alice may desire to share some of her documents with other users or applications. To this end, Alice needs to customize the sharing of her documents by adding a personalized access control (AC) policy for each user/application accessing her personal cloud. She also wants to be sure that the personal cloud is able to securely enforce the defined access controlled policies. The latter is achieved by integrating the access control engine within the secure token together with the search engine as depicted in Figure 1.

The definition of the AC policies depends on the employed AC model. Access control is a well established topic in the databases field. However, traditional AC models (e.g., Mandatory Access Control [59], Discretionary Access Control [60], Role-Based Access Control [34]) are less suitable for the Personal Cloud paradigm than tag-based access control models [42]. For instance, the work in [45] presents a study analyzing the behavior of home users (i.e., non-technical computer users) when required to define access control policies over their personal data. The study shows that the users are often technically inexperienced and often become annoyed especially with complex access control policies, leading them to take extreme decisions (i.e., to put most of the data either as private or as public). Therefore, in the context of the Personal Cloud, we need a simple AC model which basic users such as Alice can easily understand and use.

A tag-based access control (TBAC) model suits well our context where access controls rules should be fixed by non-technical users in most common cases. Moreover, a TBAC model can be easily integrated with our inverted index if we consider that (1) an appropriate set of access terms can be inserted to tag the documents at insertion, and (2) these access terms can be used at execution time to evaluate logical expressions leading to discard or not each document from the query results. Thus, in our model, an AC policy relies on a set of collection rules triggered at documents insertion to extract appropriate sets of access terms (e.g., document metadata or contextual information like date, application authoring the document or triggering the insert, etc.) and sets of access control rules defined as boolean expressions over access terms. At query time, an application has access to a document if and only if it contains sufficient access terms to validate the access control rules. We formally define below the proposed access control model.

Let D be the set of documents referred by the inverted index, as introduced in the previous section. Each document is associated with a set of terms (regular terms extracted from the documents content and access terms derived from the document content or metadata). All terms are extracted automatically from the content and metadata of the document at insertion time. Let T be the set of terms in the inverted index. For each document $d \in D$, let T_d be the set of terms that was assigned to d at insertion time. Let U be set of all users/applications that can access the Personal Cloud. Any user/application has to be authenticated before gaining access to the Personal Cloud. We assume that by default everything is private. For simplicity, we consider in the following that the access terms are derived from the document content at the time of insertion, and that applications are allowed to make query (i.e., read) only the subset of the documents in the Personal Cloud which matches the access control rules defined for that application.

DEFINITION 1.
An access control policy P_u associated to a user/application consists in the following components:

- R_u: the authorization rule for u as defined by the Cloud owner.
- *allowAccess (u, d):* a Boolean function indicating if u can or cannot read document d.

For each user $u \in U$, Alice can define an access control rule R_u (a Boolean expression of terms) defined below. Note that, if no AC rule has been defined for an application, the application cannot access any document in the Personal Cloud.

DEFINITION 2. The atomic access term-based predicates
Given a document collection D and the resulting set of access terms T, we define the term-based predicate function *tbp* as *tbp:* Tx$D \rightarrow \{0, 1\}$ such as for any $t_i \in T$ and any $d_j \in D$, $tbf(t_i, d_j) = 1$ iff $t_i \in T_{dj}$, i.e., the access term t_i appears for the document d_j, and $tbf(t_i, d_j) = 0$ otherwise. Note that the function *tbp* can be generalized, e.g., to support predicates of the form $t_i\ \theta$ value where θ can be a comparator like $>, \geq, =, \neq, \leq$.

DEFINITION 3. A specific atomic term-based predicate
Given an access term $t_i \in T$, we denote by t_i^{pred} the restriction of the access term-based predicate function *tbp* to the access term t_i, i.e., $t_i^{pred} = tbf: \{t_i\}xD \rightarrow \{0, 1\}$.

DEFINITION 4. Atomic access control rule ar_u
Given the application u, an atomic access control rule ar_u is a conjunction of specific atomic term-based predicates: $ar_u = \bigwedge_i [\neg] t_i^{pred}$

DEFINITION 5. General access control rule R_u
Given the application u, the general access control rule ar_u is a disjunction of atomic access control rules:

$$R_u = \bigvee_j (\bigwedge_i t_{i,j}^{pred})$$

Given the user/application u, the embedded access control engine has to be able to evaluate for any document d in the Personal Cloud if u has the right to access d. This is realized internally by the Boolean function *allowAccess (u, d),* which returns true iff the R_u is true on d.

Examples. Bob is a friend of Alice with whom she shares the passion for country music. Bob asks Alice to share with him her country music collection. A collection rule has been settled such that any music file inserted in the personal cloud of Alice are associated with access terms "music" and "<music_style>" where the value of music style is derived from the domain of music styles (e.g., "country", "variety", "jazz", …). Alice may define a permission rule for user Bob in her Personal Cloud, "Bob: music ∧ country", which corresponds to an atomic AC rule (see Definition 5). Thus, Bob can query all the documents in the Alice's server that contain both the access term "music" and "country". Alice has made a trip to Paris recently and Bob would like to see her trip

photos. A collection rule extracts access terms "photo", "<city>" and "<year>" attaching to each inserted photo the corresponding access terms where <city> is a city name where the photo was taken and <year> is the year when the photo was taken (would be extracted from metadata). Alice writes a new share policy for Bob, “Bob: photo ∧ Paris ∧ 2015”. The system updates the access control rule associated to the user Bob to R_{Bob} = (*music* ∧ *country*) ∨ (*photo* ∧ *Paris* ∧ *2015*) (corresponding to a general access control rule, conform to Definition 6), allowing Bob to have access to both the country music and the Paris trip photos of Alice. Alice herself needs to access her email archive over the last two years from her smartphone, but does not want that the rest of her personal data space to be accessible from her smartphone. Therefore, assuming a collection rule associated any email file with the access terms "email" and "<date>", Alice may define the rule “Alice_smartphone: (email ∧ 2014) ∨ (email ∧ 2015)”.

The proposed model presents several advantages. First, the model facilitates the usage for non-expert users [42] since it is easy to define the AC rules as showed in the above examples. Moreover, in general, the document access terms can be extracted automatically by the applications that generate or permit to manage the documents that are stored in the home server (e.g., mailer, photo or music apps). The implementation of collection rules, which is not further investigated here, can be considered as realistic enough to enable users to concentrate only on the definition of the AC policies, based on automatically extracted access terms. Finally, another important benefit is that search engine and the access control engine can be strongly integrated in a natural way since they are both built on the set of document (regular or access)-terms. This has two major consequences, which we develop in detail in the following sections: (*i*) the search and access control engines can mutualize the available resources at query execution time, which leads to lower resource requirements (which is fundamental in an embedded context), and (*ii*) the access control evaluation has negligible impact on the query and update performance.

The introduction of access control to a typical inverted index (see Chapter II) slightly changes the query evaluation. For every query $Q=\{t\}$ and a user u with an authorization rule R_u, the *tf-idf* score is computed as before by (*i*) accessing $I.S$ to retrieve for each query term t and for each term in R_u retrieve the inverted lists elements $\{I.L_t\}_{t \in Q}$; (*ii*) allocating in RAM one container for each unique document identifier in these lists; (*iii*) computing the score of each of these documents using a weight function, e.g., *tf-idf*;

(*iv*) evaluating the rule R_u for each retrieved document and updating the document score -1 if the rule is false (*v*) ranking the documents according to their score and producing the *k* documents with the highest positive scores. This general execution with our index is optimized and formalized in Chapter IV.

3.Conclusion

In this chapter, we start by introducing the state of the art methods related to the challenging problem of indexing documents in embedded systems equipped with tiny RAM and NAND Flash storage. We then formulate the problem of embedding a search engine that is RAM Bound and reaches Full scalability (i.e., for both updates and queries). The proposed design relies on three principles: (P1) Write-Once Partitioning aims at reaching insertion scalability by precluding costly random writes in the Flash; (P2) Linear Pipelining aims to evaluate the query under the RAM Bound constraint; and (P3) Background Linear Merging aims to reconcile query and update scalability. Taken together, principles P1 to P3 allow reaching the Bounded RAM and the Full scalability index properties. We also introduced in this chapter an access control mechanism closely integrated in the search engine to allow for secure document sharing. The proposed access control model relies on the current tag-based access control models given their simple way of manipulation by a common user, whice makes them the preferred models in the personal cloud paradigm. Moreover, the integration of the top-k computation with the AC rules preserves the search engine design principles. The technical solutions to implement these three principles and the AC enforcement are presented in the next chapter.

Chapter IV
Core Design of the Embedded Search and Access Control Engine

In this chapter, we present the technical solutions to implement the core of the proposed full-text search engine integrating the evaluation of tag-based access control policies on documents, based on the three principles introduced in the previous chapter, namely *Write-once Partitioning*, *Linear Pipelining* and *Background Linear Merging*. To ease the presentation, we introduce first the foundation of our solution considering only document insertions and queries. The trickier case of document deletions is postponed to Section 3. Also, we explain the access control rules' evaluation together with the query evaluation. An illustrative example, based on the personal cloud context, is presented in the next chapter and demonstrates the usability of the proposed solution.

1. Write-once Partitioning and Linear Pipelining

These two design principles are discussed together because the complexity comes from their combination. Indeed, Write-Once Partitioning is straightforward on its own. It simply consists in splitting I in a sequence $<I_0, I_1, \ldots, I_p>$ of small indexes called partitions, each one having a size bounded by RAM_Bound. The difficulty is to implement a linear pipeline execution of any query Q on this sequence of partial indexes.

For a user u with an access control rule R_u and a query Q, executing Q over I would lead to evaluate:

$$\begin{cases} Top_k\left[\sum_{t \in Q} W\left(f_{d,t}, \frac{N}{F_t}\right)\right], with\ d \in D_u \\ where\ D_u = \{\forall u \in U, \exists d \in D, allowAccess(R_u, d) = true\} \end{cases}$$

where Top_k selects the k documents $d \in D_u$ having the largest *tf-idf* scores and satisfying the access control rule R_u of the user u, each score being computed as the sum, for all query terms $t \in Q$, of a given weight function W taking as parameter the frequency $f_{d,t}$ of t in d and the inverse document frequency N/F_t. Our objective is to

remain agnostic regarding W and then let the precise form of this function open. Let us now consider how each term of this expression can be evaluated by a linear pipelining process on a sequence $<I_0, I_1, \ldots, I_p>$.

Computing N. We assume that the number of documents is a global metadata maintained at insertion/deletion time and needs not be recomputed for each Q.

Computing Ft . F_t should be computed only once for each term t since F_t is constant for Q. This is why F_t is usually materialized in the dictionary part of the index $(\{t, F_t\} \subset I.S)$, as shown in Figure 4 on page 25 . When I is split in $<I_0, I_1, \ldots, I_p>$, the global value of F_t should be computed as the sum of the local F_t of all partitions. The complexity comes from the fact that the same document d may cross several partitions with the consequence of contributing several times to the global F_t if a simple sum is performed. The Bounded RAM agreement precludes maintaining in RAM a history of all the terms already encountered for a given document d across the parsing windows, the size of this history being unbounded. Accessing the inverted lists $\{I_i.L_t\}$ of successive partitions to check whether they intersect for a given d would also violate the Linear Pipelining principle since these same lists will be accessed again when computing the *tf-idf* score of each document.

The solution is then to store in the dictionary of each partition the boundary of that partition, namely the identifiers of the first and last documents considered in the parsing window. Then, two bits *firstd* and *lastd* are added in the dictionary for each inverted list to register whether this list contains one (or both) of these documents, i.e., $\{t, F_t, firstd, lastd\} \subset I.S$. As illustrated in Figure 11, this is sufficient to detect the intersection between the inverted lists of a same term t in two successive partitions. Whether an intersection between two lists is detected, the sum of their respective F_t must be decremented by *1*. Hence, the correct global value of F_t can easily be computed without physically accessing the inverted lists.

During the F_t computation phase, the dictionary of each partition is read only once and the RAM consumption sums up to one buffer to read each dictionary, page by page, and one RAM variable to store the current value of each F_t.

Computing $f_{d,t}$. If a document d overlaps two consecutive partitions I_i and I_{i+1}, the inverted list L_t of a queried term $t \in Q$ may also overlap these two partitions. In this case the $f_{d,t}$ score of d is simply the sum of the (last) $f_{d,t}$ value in $I_i.L_t$ and the (first) $f_{d,t}$ value in $I_{i+1}.L_t$. To get the $f_{d,t}$ values, the inverted lists $I_i.L_t$ have to be accessed. The pointers

referencing these lists are actually stored in the dictionary which has already been read while computing F_t. According to the Linear pipelining principle, we avoid reading again the dictionary by storing these pointers in RAM during the F_t computation. The extra RAM consumption is minimal and bounded by the fact that the number of partitions is itself bounded thanks to the merging process (see Section 2).

Computing Top_k. Traditionally, a RAM variable is allocated to each document *d* to

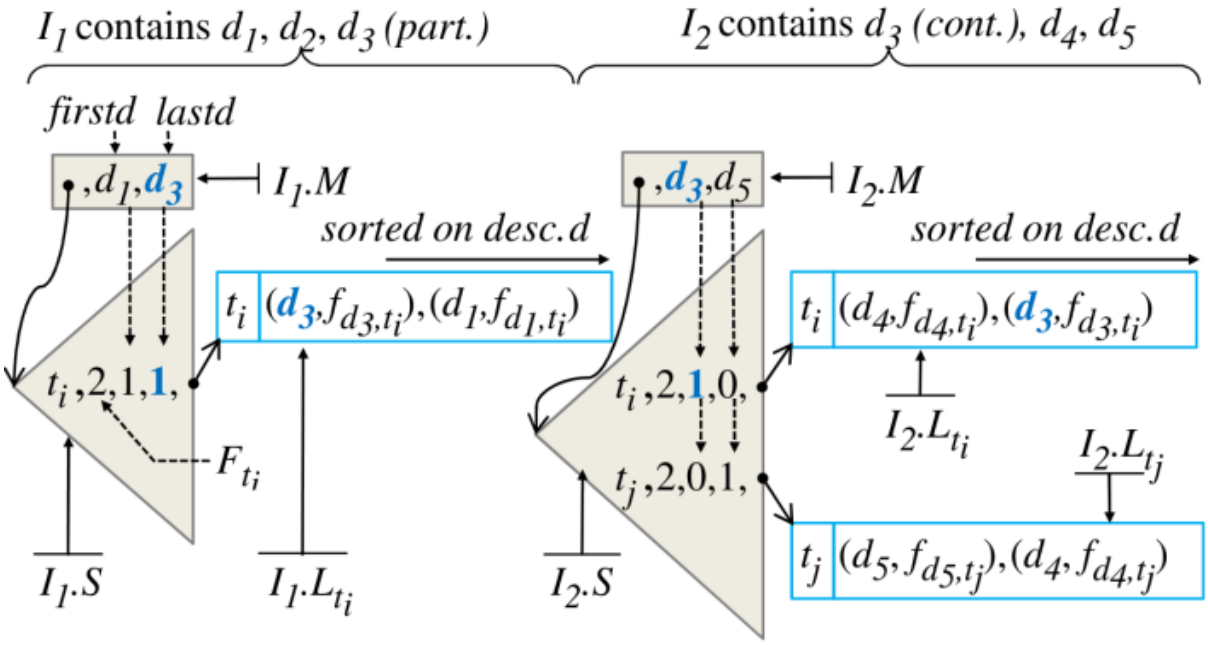


Figure 11: Consecutive index partitions with overlapping documents

compute its *tf-idf* score by summing the results of $W(f_{d,t}, N/F_t)$ for all terms $t \in Q$ [18]. Then, the *k* best scores are selected. Unfortunately, this approach conflicts with the Bounded RAM agreement since the size of the document set is likely to be much larger than the available RAM. Hence, we organize the query processing in a pure pipeline way, allocating a RAM variable only to the *k* documents having currently the best scores. This forces the complete computation of *tf-idf*(*d*) to be done for each *d*, one after the other. To meet this requirement while precluding any iteration on the inverted lists, these lists are maintained sorted on the document *id*. Note that if document *ids* reflect the insertion ordering, the inverted lists are naturally sorted. Hence, the *tf-idf* computation sums up to a simple linear pipeline merging process of the inverted lists for all terms $t \in Q$ in each partition (see Figure 12). The RAM consumption for this phase is therefore restricted to one variable for each of the current *k* best *tf-idf* scores and to one buffer (i.e., a RAM page) per query term *t* to read the corresponding inverted lists

$I_i.L_t$ (i.e., $I_i.L_t$ are read in parallel for all t, the inverted lists for the same t being read in sequence). Figure 12 summarizes the data structures maintained in RAM and in Flash to handle this computation.

Enforcing R_u. The access control rule enforcement is integrated seamlessly in the query computation. Access control rules are Boolean expressions over access terms associated to documents (e.g., authorize Bob to access all documents containing the access terms, 'vacation', '2014', 'France'). At query execution time, the evaluation of access control rules is as follows (see Figure 12). When the score of a document d is within the k best current scores, the identifier of d is searched into the inverted lists of the access terms involved in the active access rules ($T_o \in R_u$)). As soon as the evaluation Boolean expression over access terms is *false*, the document d can be discarded. If it is *true*, d is kept and inserted in the top_k. The search within the inverted lists of the access terms requires only one additional RAM page. This page is used to search for, the occurrence of d in the inverted list of each access term. This search can be performed sequentially by using a dichotomy, and stopped as soon as d has been found or not found. For evaluating conjunctive rules (made of a conjunction of terms), access terms are accessed from the less frequent access term (with the smallest inverted list) to the most frequent one, to potentially stop the search as soon as d is absent from a list (the Boolean expression is then evaluated to *false*). The overhead in terms of memory consumption for the evaluation of R_u is kept minimal (1 page) and the execution time overhead is low since (i) only the documents entering within the k best current scores trigger the evaluation of R_u, (i) the search can be performed using dichotomy if the inverted list is large (faster than a sequential search) and (iii) a subset of the access terms (and corresponding inverted lists) have to be accessed when the rule turns to be *false* (typically, for conjunctive rules). Our experiments (see Chapter VI) demonstrate the low execution time overhead due to access rule evaluation.

In our access control evaluation we consider the case of checking the access control rule after document score computation and entering the top-k list, but a clever solution can be maintained to reduce access control IO checking, by switching the computation order between the document score computation and checking the access control rule condition, in the lower level the document score is less selective than the access control checking, so in the begging of the query evaluation (in the lower level of the index), starting by checking the access control rule can be more effective in IO cost

(the document score computation enter the top-k frequently), in the higher levels of the index, we have longest inverted lists, starting the evaluation by access control rule checking can reduce the number of IOs (see Figure 12).

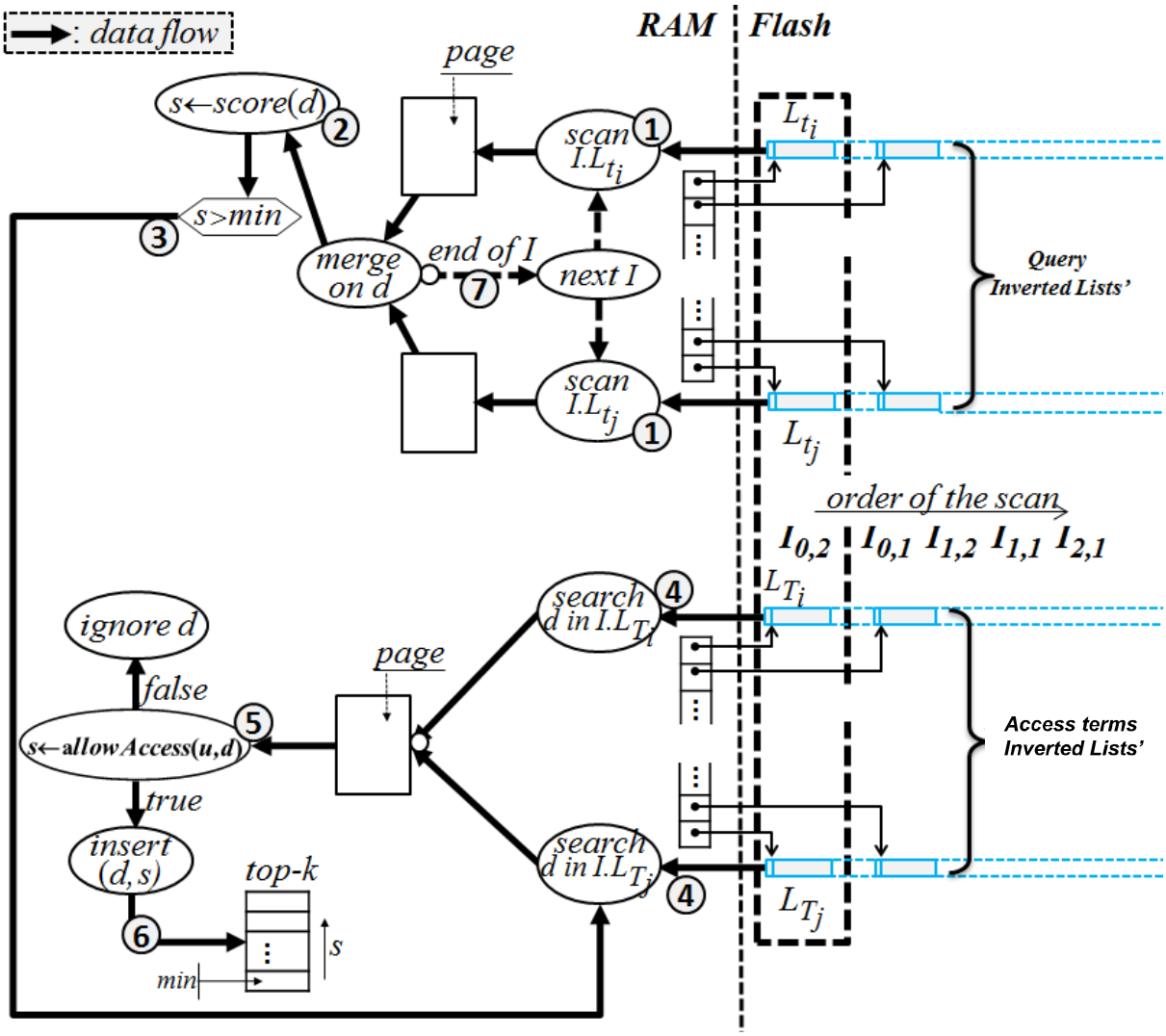


Figure 12: Linear Pipeline computation of Q over terms t_i and t_j and access terms L_{Ti} and L_{Tj}

2. Background Linear Merging

The background merging process aims at achieving scalable query costs by timely merging several small indexes into a larger index structure. As mentioned in chapter 3, the merge must be a pipeline process in order to comply with the Bounded RAM agreement while keeping a cost compatible with the update rate. Moreover, the query processing should continue to be executed in Linear Pipelining (see Section 1) on the

structure resulting from the successive merges. Therefore, the merges have to preserve the global ordering of the document *ids* within the index structures.

To meet these requirements, we introduce a Sequential and Scalable Flash structure, called *SSF*, pictured in Figure 13. The *SSF* consists in a hierarchy of partitions of exponentially increasing size. Specifically, each new index partition is flushed from RAM into the first level of the *SSF*, i.e., L_0. The *merge* operation is triggered automatically when the number of partitions in a level becomes *b*, the branching factor of *SSF*, which is a predefined index parameter. The merge combines the *b* partitions at level L_i of SSF, denoted by $I_1^i,...I_b^i$, into a new partition at level L_{i+1}, denoted by I_j^{i+1} and then reclaims all partitions at level L_i.

The *merge* is directly processed in pipeline as a multi-way merge of all partitions at the same level. This is possible since the dictionaries of all the partitions are already sorted on terms, while the inverted lists in each partition are also sorted on document ids. So are the dictionary and the inverted lists of the resulting partition at the upper level. More precisely, the algorithm works in two steps. In the first step, the *I.L* part of the output partition is produced. Given *b* partitions in the index level L_i, *b*+1 RAM pages are necessary to process the merge in linear pipeline: *b* pages to merge the inverted lists in *I.L* of all *b* partitions and one page to produce the output. The indexed terms are treated one after the other in alphabetic order. For each term *t*, the head of its inverted lists in each partition is loaded in RAM. These lists are then consumed in pipeline by a multi-way merge. Document ids are encountered in descending order in each list and the output list resulting from the merge is produced in the same order. A particular case must be distinguished when two pairs (*d*, $f1_{d,t}$) and (*d*, $f2_{d,t}$) are encountered in separate lists for the same *d*; this means that document *d* overlaps two partitions and these two pairs are aggregated in a single (*d*, $f1_{d,t} + f2_{d,t}$) before being added to *I.L*. In the second step, the metadata *I.M* is produced (see Figure 11), by setting the value of *firstd* (resp. *lastd*) with the *firstd* (resp. *lastd*) value of the *first* (resp. *last*) partition to be merged, and the *I.S* structure is constructed sequentially, with an additional scan of *I.L*. The *I.S* tree is built from the leaves to the root. This step requires one RAM page to scan *I.L*, plus one RAM page per *I.S* tree level. For each list encountered in *I.L*, a new entry (*t*, F_t, *presence_flags*) is appended to the lowest level of *I.S*; the value F_t is obtained by summing the $f_{d,t}$ fields of all (*d*, $f_{d,t}$) pairs in this list; the presence flag reflects the presence in the list of the *firstd* or *lastd* document. Upper levels of *I.S* are then trivially filled sequentially. This Background Merging process generates only sequential writes

in Flash and previous partitions are reclaimed in large blocks after the merge. This pipeline process sequentially scans each partition only once and produces the resulting partition also sequentially. Hence, assuming *b*+1 is strictly lower than RAM_bound, one RAM buffer (of one page) can be allocated to read each partition and the merge is I/O optimal. If *b* is larger than RAM_bound, the algorithm remains unchanged but its I/O cost increases since each partition will be read by page fragments rather than by full pages.

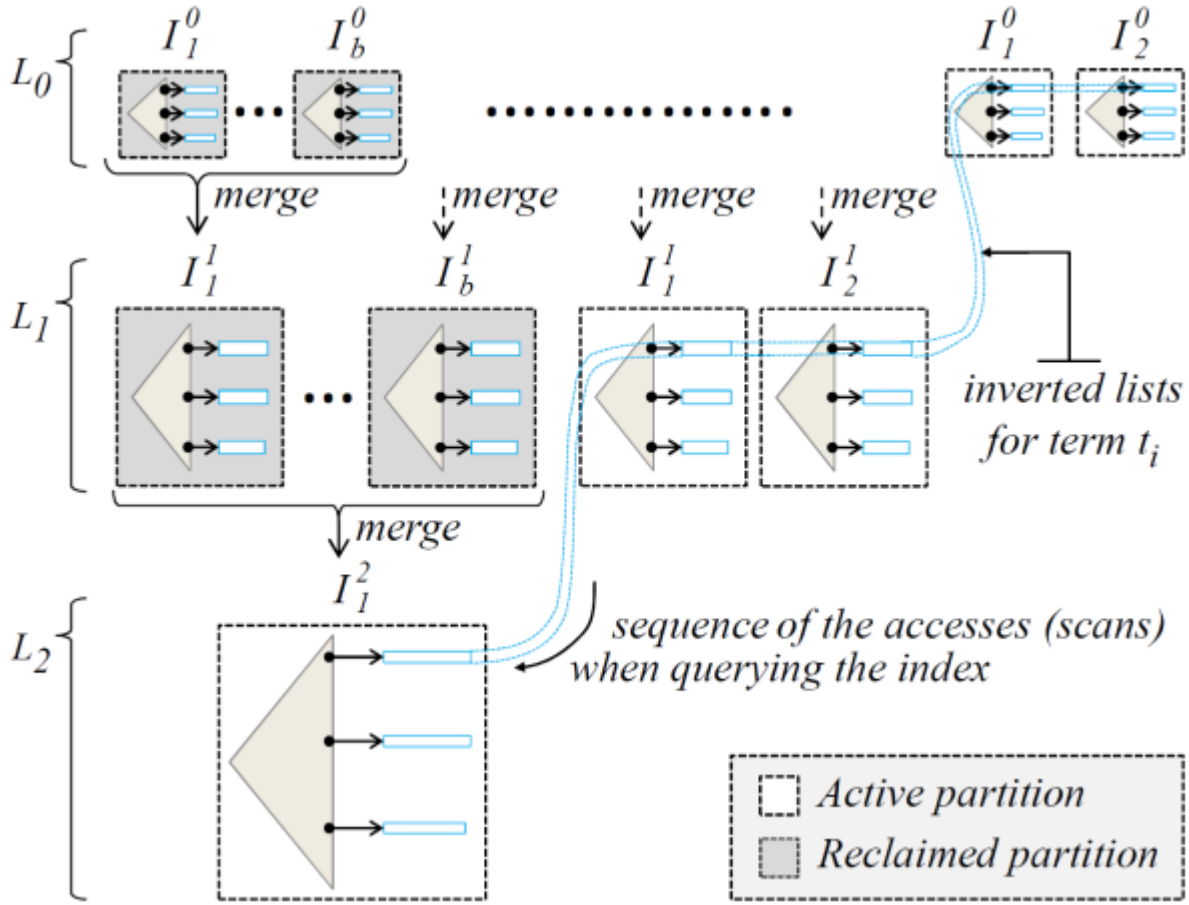


Figure 13: The Scalable and Sequential Flash structure

Search queries can be evaluated in linear pipeline by accessing the partitions one after the other from partitions *b* to 1 in level 1 up to level *n*. In this way, the inverted lists are scanned in descending order of the document ids, from the most recently inserted document to the oldest one, and the query processing remains exactly the same as the one presented in Section 1, with the same RAM consumption. The merging and the querying processes could be organized in opposite order (i.e., in ascending order of the document ids) with no impact. However, order matters as soon as deletions are considered (see Section 3). *SSF* provides scalable query costs since the amount of

indexed documents grows exponentially with the number of levels, while the number of partitions increases only linearly with the number of levels.
Note that merges in the upper levels are exponentially rare (one merge in level L_i for b^i merges in L_0) but also exponentially costly. To mitigate this problem, we perform the merge operations in background (i.e., in a non-blocking manner). Since the merge may consume up to b pages of RAM, we launch/resume it each time after a new partition is flushed in L_0 of the *SSF*, the RAM being empty at this time. A small quantum of time (a few hundred milliseconds in practice) is allocated to the merging process. Each time this quantum expires, the merge is interrupted and its execution status (i.e., a cursor indicating the current Flash page position in each partition) is memorized. The quantum of time is chosen so that the merge of a given *SSF* level ends before the next merge of the same level needs to be triggered. In this way, the cost of a merge operation is spread among the flush operations and remains almost transparent. This basic strategy is simple and does not make any assumption regarding the index workload. However, it could be improved in certain contexts, by taking advantage of the idle time of the platform.

3. Document Deletions

To the best of our knowledge, our proposal is the first embedded search index to implement document deletions. This problem is actually of primary importance because deletions are required in many practical scenarios. Unfortunately, index updating increases significantly the complexity of the index maintenance by reintroducing the need for random updates in the index structure. In this section we extend the index structure to support the deletions of documents without generating any random write in Flash.

3.1 Solution Outline

Implementing the delete operation is challenging, mainly because of the Flash memory constraints which proscribe the straightforward approach of updating in-place the inverted index. The alternative to updating in-place is *compensation*, i.e., the deleted documents' identifiers (*DDIs*) are stored in an appropriate way and used as a filter to eliminate the ghost documents retrieved by the query evaluation process.

A basic solution could be to organize the *DDIs* as a sorted list in Flash and to intersect this list at query execution time with the inverted lists in the *SSF* corresponding to the

query terms. However, this solution raises several problems. First, the documents are deleted in random order, according to users' and application decisions. Hence, maintaining a sorted list of *DDIs* in Flash would violate the Write-Once Partitioning principle since the list has to be rewritten each time a set (e.g., a page) of new *DDIs* is flushed from RAM. Second, the computation of the F_t for each query term t during the first step of the query processing cannot longer be achieved without an additional merge operation to subtract the sorted list of *DDIs* from the inverted lists of the *SSF*. Third, the full *DDI* list has to be scanned for each query regardless of the query selectivity. These two last elements make the query cost dependent on the total number of deleted documents and then conflict with the Linear pipelining principle.

Therefore, instead of compensating the query evaluation process, we propose a solution based on compensating the indexing structure itself. In particular, a document deletion is treated similarly to a document insertion, i.e., by re-inserting the metadata (terms and frequencies) of all deleted documents in the *SSF*. The objective is threefold: (*i*) to be able to compute, as presented in Section 1, the F_t for each term t of a query based on the metadata only (of both existing and deleted documents), (*ii*) to have a query performance that depends on the query selectivity (i.e., number of inserted and deleted documents relevant to the query) and not on the total number of deleted documents and (*iii*) to effectively purge the indexing structure from the largest part of the deleted documents at Background Merging time, while remaining compliant with the Linear Pipelining principle. We present in the following the required modifications of the index structure to integrate this form of compensation.

3.2 Impact on Write-Once Partitioning

As indicated above, a document deletion is treated similarly to a document insertion. Assuming a document d is deleted in the time window corresponding to a partition I_i, a pair (d, $-f_{d,t}$) is inserted in each list $I_i.L_t$ for the terms t present in d and the F_t value associated to t is decremented by 1 to compensate the prior insertion of that document. To distinguish between an insertion and a deletion, the frequency value $f_{d,t}$ for the deleted document *id* is simply stored as a negative value, i.e., $-f_{d,t}$.

3.3 Impact on Linear Pipelining

Executing a query Q over our compensated index structure sums up to evaluate:

$$\begin{cases} Top_k\left[\sum_{t\in Q} W\left(|f_{d,t}|, \frac{N}{F_t}\right)\right], with\ d \in (D_u^+ - D^-) \\ where\ D_u^+ = \{\exists u\ \in U, \exists d \in D, allowAccess(u,d) = true\} \end{cases}$$

where D^+ (resp. D^-) represents the set of inserted (resp. deleted) documents.

Computing N. As presented earlier, *N* is a global metadata maintained at update time and then already integrates all insert and delete operations.

Computing F_t **.** The global F_t value for a query term *t* is computed as usual since the local F_t values are compensated at deletion time (see above). The case of deleted documents that overlap with several consecutive partitions is equally treated as with the inserted documents.

Computing $f_{d,t}$ **.** The $f_{d,t}$ of a document *d* for a term *t* is computed as usual, with the salient difference that a document which has been deleted appears twice: with the value (*d*, $f_{d,t}$) (resp. (*d*, -$f_{d,t}$)) in the inverted lists of the partition l_i (resp. partition l_j) where it has been inserted (resp. deleted). By construction $i < j$ since a document cannot be deleted before being inserted.

Computing Top_k**.** Integrating deleted documents makes the computation of Top_k more subtle. Following the Linear Pipelining principle, the *tf-idf* scores of all documents are computed one after the other, in descending order of the document ids, thanks to a linear pipeline merging of the insert lists associated to the queried terms. To this end, the algorithm introduced in Section 1 uses *k* RAM variables to maintain the current *k* best *tf-idf* scores and one buffer (i.e., a RAM page) per query term *t* to read the corresponding inverted lists. Some elements present in the inverted lists correspond actually to deleted documents and must be filtered out. The problem comes from the fact that documents are deleted in random order. Hence, while inverted lists are sorted with respect to the insertion order of documents, a pair of the form (*d*, -$f_{d,t}$) may appear anywhere in the lists. In case a document *d* has been deleted, the unique guarantee is to encounter the pair (*d*, -$f_{d,t}$) before the pair (*d*, $f_{d,t}$) if the traversal of the lists follows a descending order of the document ids. However, maintaining in RAM the list of all encountered deleted documents in order to filter them out during the follow-up of the query processing would violate the Bounded RAM agreement.

The proposed solution works as follows. The *tf-idf* score of each document *d* is computed by considering the modulus of the frequencies values $|\pm f_{d,t}|$ in the *tf-idf* score computation, regardless of whether *d* is a deleted document or not. Two lists are maintained in RAM: Top_k = {(*d*, *score*(*d*))} contains the current *k* best *tf-idf* scores of documents which exist with certainty (no deletion has been encountered for these documents); *Ghost* = {(*d*, *score*(*d*))} contains the list of documents which have been deleted (a pair (*d*, $-f_{d,t}$) has been encountered while scanning the inverted lists) and have a score better than the smallest score in Top_k. Top_k and *Ghost* lists are managed as follows. If the score of the current document *d* is worse than the smallest score in Top_k, it is simply discarded and the next document is considered (step 2 in Figure 14). Otherwise, two cases must be distinguished. If *d* is a deleted document (a pair (*d*, $-f_{d,t}$) is encountered), then it enters the *Ghost* list (step 3); else it enters the Top_k list unless its id is already present in the *Ghost* list (step 4). Note that this latter case may occur only if the id of *d* is smaller than the largest *id* in *Ghost*, making the search in *Ghost* useless in many cases. An important remark is that the *Ghost* list has to register only the deleted documents which may compete with the *k* best documents, to filter them out when these documents are later encountered, which makes this list very small in practice.

While simple in its principle, this algorithm deserves a deeper discussion in order to evaluate its real cost. This cost actually depends on whether the *Ghost* list can entirely reside in RAM or not. Let us compute the nominal size of this list in the case where the deletions are evenly distributed among the document set. For illustration purpose, let us assume *k*=10 and the percentage of deleted documents δ=10%. Among the first 11 documents encountered during the query processing, 10 will enter the Top_k list and 1 is likely to enter the *Ghost* list. Among the next 11 documents, 1 is likely to be deleted but the probability that its score is in the 10 best scores is roughly 1/2. Among the next 11 ones, this probability falls to about 1/3 and so on and so forth. Hence, the nominal size of the *Ghost* list is $\delta \cdot k \cdot \sum_{i=1}^{n} 1/i$, which can be approximated by $\delta \cdot k \cdot (\ln(n) + \varepsilon)$. For 10.000 queried documents, *n*=1000 and the size of the *Ghost* list is only $\delta \cdot k \cdot (\ln(n) + \varepsilon) \approx 10$ elements, far beyond the RAM size. In addition, the probability that the score of a *Ghost* list element competes with the Top_k ones decreases over time, giving the opportunity to continuously purge the *Ghost* list (step 5 in Figure 14). In the very improbable case where the *Ghost* list overflows (step 6 in Figure 14), it is sorted in descending order of the document ids, and the entries corresponding to low

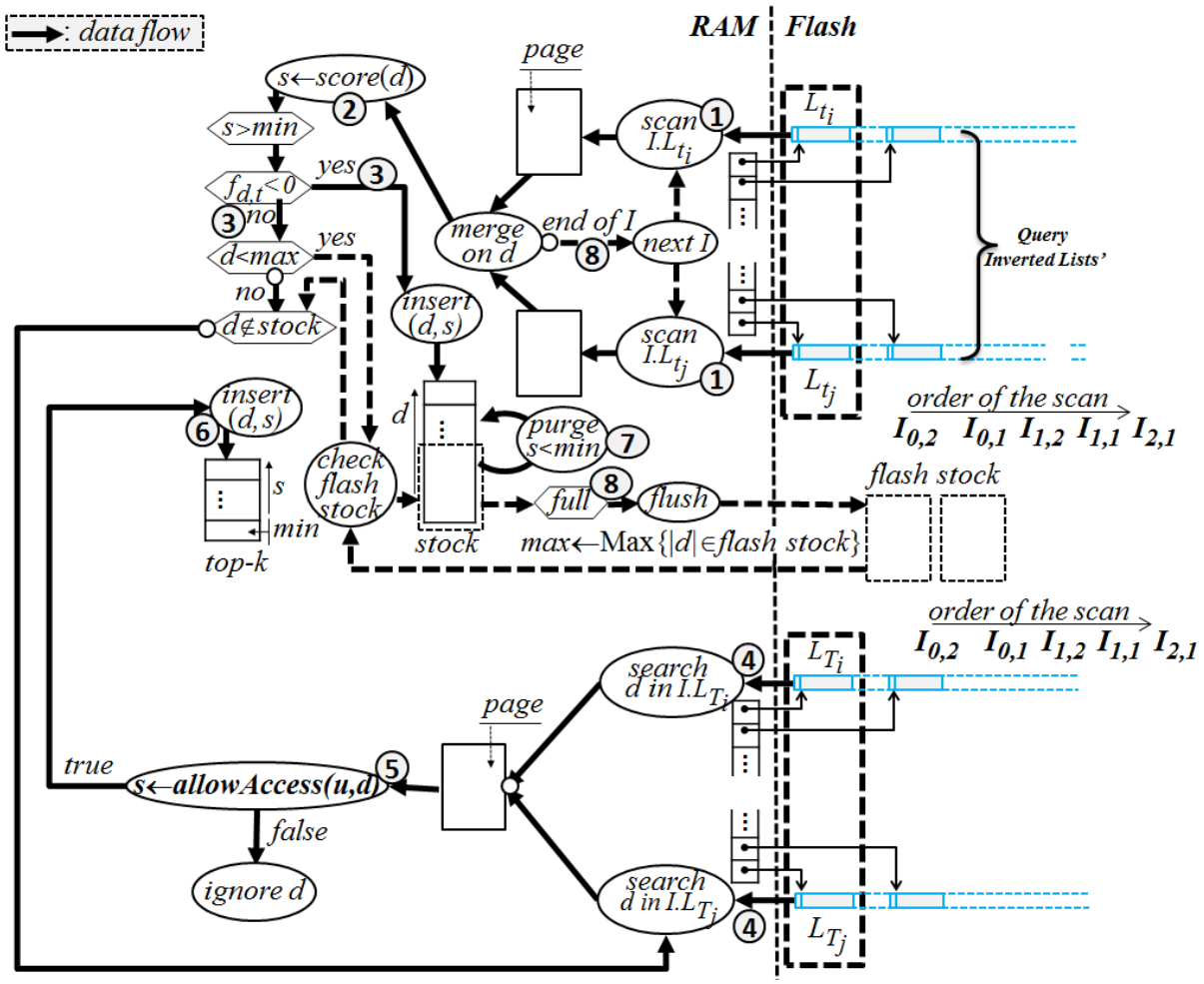


Figure 14: Linear pipeline computation of *Q* in the presence of deletions

document ids are flushed. This situation remains however highly improbable and will concern rather unusual queries (none of the 300 queries we evaluated in our experiment produced this situation, while allocating a single RAM page for the *Ghost* list).

Enforcing R_u. The enforcement of access control rules with delete is integrated in the query computation as presented in Section 1, by allocating one additional page to read access terms' inverted lists involved in the access control rules ($T_o \in R_u$) for each document within the k best current scores and only if the document is not considered as a deleted document. Deleted documents can be inserted in the *Ghost* list without checking the access control rules to minimize access control IO cost). This process is illustrated in Figure 14.

3.4 Impact on Background Pipeline Merging

The main purpose of the Background Merging principle, as presented in Section 2, is to keep the query processing scalable with the indexed collection size. The introduction of deletions has actually a marginal impact on the merge operation, which continues to be efficiently processed in linear pipeline as before. Moreover, given the way the deletions are processed in our structure, i.e., by storing couples (d, $-f_{d,t}$) for the deleted documents, the merge acquires a second function which is to absorb the part of the deletions that concern the documents present in the partitions that are merged. Indeed, let us come back to the Background Merging process described in Section 2. The main difference when deletes are considered is the following. When inverted lists are merged during step 1 of the algorithm, a new particular case may occur, that is when two pairs (d, $f_{d,t}$) and (d, $-f_{d,t}$) are encountered in separate lists for the same d; this means that document d has actually been deleted; d is then purged (the document deletion is absorbed) and will not appear in the output partition. Hence, the more frequent the Background Merging, the smaller the number of deleted entries in the index.

Taking into account the supplementary function of the merge, i.e., to absorb the data deletions, we can adjust the absorption rate of deletions by tuning the branching factor of the last index level since most of the data is stored in this index level. By setting a smaller value to the branching factor b' of the last level, the merge frequency in this level increases and consequently the absorption rate also increases. Therefore, in our implementation we use a smaller value for the branching factor of the last index level (i.e., b'=3 for the last level and b=10 for the other levels). Typically, about half of the total number of deletions will be absorbed for b'=3 if we consider that the deletions are uniformly distributed over the data insertions.

5. Conclusion

This chapter details the core design of the embedded search and access control engine for hardware-constrained devices equipped with tiny RAM and NAND Flash storage. The chapter starts with the implementation of the first two design principles, Write-once Partitioning and Linear Pipelining, given their interdependence by considering only document insertions and queries. Then, the implementation of the third principle, Background Linear Merging, is presented and its role to reduce the query cost is explained. Later in this chapter, we discuss the tricky case of document

deletion and show its implementation in accordance with the three design principles. In each of the above steps, the evaluation of access control rules is discussed separately to show the double functionality of the embedded search engine, i.e., for document search and access control policy enforcement. The proposed solution has been published in the VLDB conference [61].

The following chapter presents an illustrative use case for the embedded search and an access control engine, to demonstrate the usability of the proposed solution.

Chapter V
Implementation Issues

In this chapter, we present the aspects related to the implementation issues of the proposed search engine. We start by recalling our objectives and scenarios that motivate the search engine. Then, we give a brief description of the ANR KISS[17] project and the PlugDB[18] engine [62], since this work is strongly related to both, and indicate our contribution in the KISS project. We then detail all the algorithms of the SSF structure proposed in the previous chapter. Finally, we present our prototype platform and describe how we can demonstrate the properties of the proposed architecture (i.e., scalability, RAM Bound and security), through a scenario illustrating the Personal Cloud paradigm [63].

1. Objectives and Scenarios

With the convergence of mobile communications, sensors and online social networks technologies, we are witnessing an exponential increase in the acquisition of personal data.

The Personal Cloud paradigm emerges as a decentralized and privacy preserving solution to manage personal documents under the users' control. It can be thought of as a dedicated box connected to the user's internet gateway or a pluggable computer, equipped with storage, computing and communication facilities, running a personal server and acquiring data from multiple sources (such as smartphones, cameras, banks, web sites, employer, doctors).

The documents entering a user dataspace can be any form of files (pictures, text files, pdf files, mails, data steams produced by sensors, etc.). To query such personal data. We considered the use of a simple keyword search engine, in the spirit of Google desktop or Spotlight. Terms are extracted from the files content and from metadata describing them (e.g., name, type, date, creator, tags set by the user herself). Since keywords seem the most convenient way to query the user dataspace, it makes sense

[17] https://project.inria.fr/kiss/en/
[18] https://project.inria.fr/plugdb/

to let users express access control rules with a similar paradigm, that is through logical expressions on terms associated to documents.
However, letting unexperienced users managing all these personal data exposes them to major security breaches. To make this architecture secure, we propose to employ a secure co-server to manage all the sensitive data (e.g., cryptographic keys, metadata, indexes) and data processing (e.g., query evaluation and access control). Hence, user personal data can be stored encrypted in a personal computer or the Cloud, but the metadata, the index to query this data and the encryption keys are stored in the secure co-server. In our case, this co-server plays the role of a secure Google desktop/Spotlight for the user's personal data. The high level of security and trust of this architecture is rooted in the combination of hardware and software security and data decentralization.

2. KISS Project

The work in this thesis is related to the ANR KISS (Keep your personal Information Safe and Secure) project, which proposes the implementation of a secure and portable Personal Data Server (see ANR KISS grant n° ANR-11-INSE-0005). Therefore, in this chapter, we present an overview of the KISS project [64] and describe the integration of our contributions in this project.
The exponential production of personal data in the last years has been accompanied by the massive concentration of the data on servers by administrations, hospitals, insurance companies, etc. Meanwhile, an increasing amount of digitized personal data is sent to citizens (salary forms, insurance forms, invoices, phone call sheets, banking statements, etc), who themselves often count on service providers to reliably store this data and make it available through the Cloud. However, these benefits must be weighed against privacy risks incurred by centralizing data on servers. Indeed, there are many examples of privacy violations arising from negligence, abusive use or attacks, and even the most secured servers are not spared. The KISS project draws a radically different vision of the management of personal data. It builds upon the emergence of new portable and secure devices known as secure tokens (e.g., mass storage SIM cards, secure USB sticks, smart sensors) combining the security of smart cards and the storage capacity of NAND Flash chips. The idea promoted in KISS is to embed, in such devices, software components capable of acquiring, storing and managing securely personal data. These software components form a full-fledged Personal Data Server (PDS) which can remain under the holder's control. However,

the approach does not sum up to a simple secure repository of personal data. The ambition is threefold. The first objective is to allow the development of new, powerful, user-centric applications thus requiring a well-organized, structured and queryable representation of user's data. Second, KISS wants to provide the data holder with a friendly control over the sharing conditions related to her data and to provide the data recipient with certified information related to their provenance. Third, to give sense to this vision, PDSs must provide traditional database services like durability, query facilities, transactions and must be able to interoperate with external data sources.
Compared to an approach where all personal data is gathered on traditional servers, the benefit provided by PDS is manifold: (1) the PDS holder becomes his own Storage Provider thereby precluding abusive usages from external service providers, (2) secure tokens provides tangible elements of trust (i.e., tamper-resistance, holder's ownership) which cannot be provided by any traditional server, (3) privacy principles can be enforced for the data externalized by the holder provided the recipient of this data is another PDS and (4) the holder's data remains available in disconnected mode.
Converting the Personal Data Server vision into reality introduces however several scientific challenges. The secure token, central element of the approach, exhibits strong hardware constraints (e.g., little RAM, NAND Flash storage). Traditional core database techniques (storage and indexing, query and transaction processing) need then to be fully revisited to design an embedded database engine that provides acceptable performance whatever the form of the embedded data (regular, documents or spatio-temporal) and of the queries (SQL-like, full-text search, spatio-temporal search).
Figure 15 shows the abstract embedded architecture of a PDS. To implement this architecture, the KISS project relies on PlugDB[19]. PlugDB is an embedded database engine responsible for organizing personal data in a relational database style, indexing the data, executing queries on it and protecting it through access control rules and encryption of the data at rest. However, PlugDB currently implements only a small part of the KISS architecture, focusing mainly on relational data. Therefore, we need to extend PlugDB with additional functionalities required by the PDS architecture. In particular, the work in this thesis considers the management of documents (or files) in the PDS architecture. Our contributions in this architecture are highlighted in red in

[19] https://project.inria.fr/plugdb/

Figure 15 and mainly focus on the implementation of the embedded search engine to index the documents and evaluate search queries and TBAC policies on the documents.

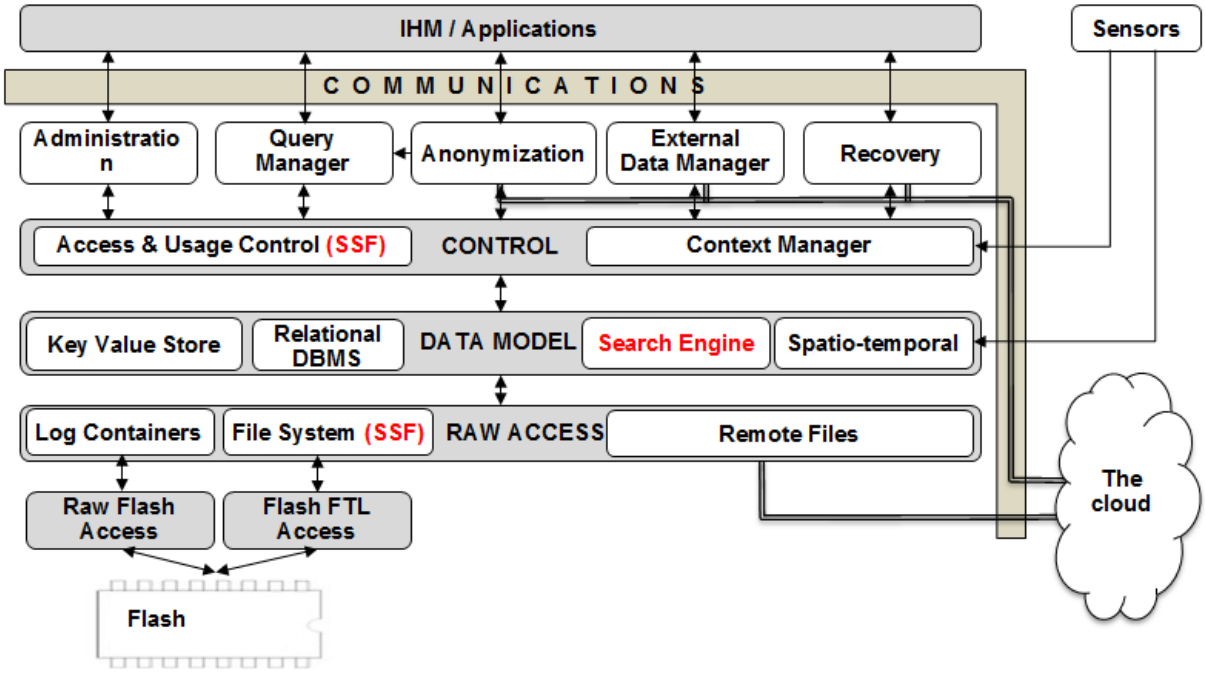


Figure 15: KISS Personal Data Server Architecture

The interested reader can find a more detailed description of the KISS project in [64]. In the following subsections, we focus on specific aspects of the KISS architecture related to document search and sharing. Also, our extension of PlugDB [62] has been demonstrated in [63], as detailed in Section 4 below.

3. Algorithms

In this section, we provide the detailed algorithm pseudocode of insertion/deletion (Algorithm 1), merge (Algorithm 2) and search (Algorithm 3). Based on these algorithms, we rediscuss in more detail the SSF operations and also the exact memory consumption of each operation to show that, by construction, all the operations respect the RAM_Bound agreement (as explained in chapter III and IV).

Algorithm 1 presents the pseudocode of the insertion and deletion operations. Since a deletion is treated as an insertion by the SSF, the algorithm is the same for both operations. The only difference is that for an insertion the frequency f_t of each document term is a positive value, whereas for a deletion the term frequencies f_t are negative values. An inserted/deleted document is represented in the index by a set of

triples *{(t, f_t, d)}* with *t* a (distinct) term and f_t its frequency in a document and d the identifier of that document. Tags are considered as terms at the insertion time. Each triple is initially in RAM (line 19 in Algorithm 1) until the buffered RAM partition reaches the RAM_Bound limit. Then the RAM partition is written in Flash and its inverted lists are indexed on the terms with a non-dense B+-tree (see line 6 in Algorithm 1 and also Algorithm 4). After the RAM flush, a merge (see Algorithm 2) is triggered if the number of partitions in the level 0 of the SSF reaches the branching factor b. The algorithm also checks if other merges are required in the upper SSF levels (lines 10 to 16 in Algorithm 1), since a merge in an SSF level creates a new partition in the subsequent level. By construction, the memory consumption of Algorithm 1 is limited to the RAM_Bound. The memory consumption of the algorithms to build the hierarchical index and to merge is discussed below.

Algorithm 1. SSF Insertion / Deletion

Input: A set D of triples $\{(t, f_t, d)\}$ with t a (distinct) term and f_t its frequency in a document and d the identifier of that document.

Output: ∅.

```
1.  for each triple e ∈ D do
2.    if sizeof(partition in RAM) == RAM_Bound then  /* Flush the RAM partition in Flash */
3.      write {t, f_t, {(d, f_{d,t})}↓d}↓t from the RAM partition to a chained list of Flash sectors;
4.      ptr = address of the first written Flash sector of the new partition;
5.      i = smallest index i | P[0][i] = ∅; /* this smallest i always exists here and i ∈ [1, b] */
        /* Build the hierarchical index and return the address of the root sector */
6.      root = Build_hierachical_index(ptr); /* see Algorithm 4 */
7.      P[0][i] = root;
8.      if (i == b) /* Merge the partitions if the number of partitions in the level reaches the
    branching factor b */
9.        l = 0;
10.       while P[l][p] ≠ ∅, ∀p ∈ [1, b] do  /* while level l contains b partitions */
11.         Merge(l); /* see Algorithm 2 */
12.         for p = 1 to k do /* free all partitions in level l */
13.           P[l][p] = ∅;
14.         end
15.         l++;
16.       end
17.   end
18. end
19. insert the triple e in the RAM partition;
```

20. **End**

Algorithm 2 presents the pseudocode for the merge operation. The merge is directly processed in pipeline as a multi-way merge of all partitions at the same level. This is possible since the dictionaries of all the partitions are already sorted on terms, while the inverted lists in each partition are also sorted on document ids. So are the dictionary and the inverted lists of the resulting partition at the upper level. More precisely, the algorithm works in two steps. In the first step, the *I.L* part (i.e., the set of inverted lists) of the output partition is produced (lines 6 to 27 in Algorithm 2). Given *b* partitions in the index level L_i, *b+1* RAM pages are necessary to process the merge in linear pipeline: *b* pages to merge the inverted lists in *I.L* of all *b* partitions (line 1 in Algorithm 2) and one page to produce the output (line 2 in Algorithm 2). The indexed terms are treated one after the other in alphabetic order (line 6 in Algorithm 2). For each term *t*, the head of its inverted lists in each partition is loaded in RAM (line 10 in Algorithm 2). These lists are then consumed in pipeline by a multi-way merge (lines 21 and 22 in Algorithm 2). Document ids are encountered in descending order in each list and the output list resulting from the merge is produced in the same order. For the *b* partition pages loaded in RAM, a border term is computed (line 14 in Algorithm 2). Then, all the terms inferior to the border term can be safely merged and their inverted lists written in the new partition. The border term is updated whenever a new partition page is loaded in RAM (line 10 in Algorithm 2). In the second step (line 29 in Algorithm 2), the *I.S* structure is constructed sequentially, with an additional scan of *I.L* (see Algorithm 4). This Background Merging process generates only sequential writes in Flash and previous partitions are reclaimed in large blocks after the merge. This pipeline process sequentially scans each partition only once and produces the resulting partition also sequentially. Hence, assuming *b*+1 is strictly lower than RAM_bound, one RAM buffer (of one page) can be allocated to read each partition and the merge is I/O optimal. If *b* is larger than RAM_bound, the algorithm remains unchanged but its I/O cost increases since each partition will be read by page fragments rather than by full pages. Hence, the memory consumption of the merge operation will be lower than the RAM_Bound in all cases. The merge algorithm also requires storing *b*+1 pointers in RAM. However, the RAM consumption for these variables represents only a fraction of a RAM page and is negligible compared to the *b*+1 RAM pages required by the multi-way merge.

Algorithm 2. SSF Merge

Input: *l* level of the SSF with the partitions to be merged.

Output: ∅.

1. *In*[*b*] an array of *b* sectors allocated in RAM ; /* *b* RAM pages to concomitantly read the *b* partitions in level *l* */
2. *Out* a sector allocated in RAM ; /* one RAM page to temporarily buffer the partial result of the merge */
3. *Ptr* the address of a Flash sector ;
4. *Ptr*[*b*] an array of *b* addresses of Flash sectors ;
5. memset(*Out*, ∅); /* *initialize Out* */
6. $\forall x \in [1, b]$, *Ptr*[*x*] = Access_hierarchical_index (*P*[*l*][*x*], *-1 /* lowest term */*) ; /* *initialize Ptr*[] *with the first Flash sector of each partition* */
   /* *Perform in pipeline the Multi-way merge* */
7. **while** $\exists$ *Ptr*[*x*] ≠ ∅ | $x \in [1, b]$ **do**
8. **for** *x* = 1 to *b* **do**
9. **if** *Ptr*[*x*] ≠ ∅ and *Ram*[*x*] *is empty* **then**
10. *In*[*x*] = load Flash sector *Ptr*[*x*] ; /* *In*[*x*] ⊃ elements of $\{t, f_t, \{(d, f_{d,t})\}\downarrow_d\}\downarrow_t$ */;
11. *Ptr*[*x*] = *Ptr*[*x*] + sizeof(Flash_sector); /* *address of the next partition sector in Flash* */
12. **end**
13. **end**
14. $t_{frontier}$ = min ({ $\forall x \in [1, b]$, max ({ $t \in In[x]$ }) }) ; /* *find the border term for all the terms in RAM, i.e., all the terms inferior to the border term can be safely merged* */
15. **while** $\exists$ $x \in [1, b]$ and $t \in In[x]$ | $t \leq t_{min}$ **do**
16. **if** *Out* is full **then**
17. write *Out* in the next free Flash sector ;
18. memset(*Out*, ∅); /* *Delete content of Out* */
19. **end**
20. t_{next} = min ({ $t \in In[x]$ | $x \in [1, b]$ }) ;
21. *E* = { elements e_x of type $<t, f_t, \{(d, f_{d,t})\}>$ | $e_x \in In[x]$, $e_x.t = t_{next}$, $x \in [1, b]$ } ;
22. write in *Out* the element $< t_{next}$, $\sum_{1 \leq x \leq b} e_x.f_t$, { $e_1.\{(d, f_{d,t})\}$ + ... + $e_b.\{(d, f_{d,t})\}$ } > ; /* *the resulted list of* t_{next} *the concatenation of all the partial lists of* t_{next} *in the merged partitions* */
23. remove all the elements $e_x \in E$ from *In* ;
24. **end**
25. write *Out* in the next free Flash sector ;
26. memset(*Out*, ∅);
27. **end**
28. free (*In*); free(*Out*);
29. *root* = Build_hierarchical_index (*Ptr*) ; /* *index the terms of the newly created partition* */
21. *p* = smallest index *p* | *P*[*l+1*][*p*] = ∅;
22. *P*[*l+1*][*p*] = *root* ; /* *store in the index metadata the root of the index of the new partition* */

23.**end**

Algorithm 3 presents the query processing algorithm in the SSF. Given a set of query terms with an authorization rule R_u for the user u and an integer value k, the algorithm returns an array of k couples *(d, tfidf_score)* of document identifiers and their *tf-idf* score satisfying the authorization rule R_u. The search algorithm consists in two steps. First, the F_t value of each query term is computed (line 3 in Algorithm 3). This part is described in detail in Algorithm 5. Second, the list of top-k documents with the highest scores is obtained (line 4 in Algorithm 3). This part is described in detail in Algorithm 6.

Algorithm 3. SSF Search

Input: $Q = \{q_i\}$ a set of q query terms; $R_u=\{To_i\}$ a user u authorization rule; k the requested number of results (top-k).

Output: $R[k]$ an array of k couples (d, *tfidf_score*) of document identifiers and their *tf-idf* score.

1. $Ft[q]$ an array of q values to store the Ft frequency for each query term initialized at 0 ;
2. $Ptr[q][l][b]$ a set of pointers to the start of the inverted lists of each query term in each partition in each index level;
3. Ptr = Compute_Ft (Q, Ft) ;/* *compute the F_t for each query term, see Algorithm 5* */
4. R = Compute_Top_k (Q, Ft, Ptr, k, R_u) ; /* *compute the top-k results (in pipeline), see Algorithm 6* */
5. **return** R;

Algorithm 4 presents the pseudocode of the construction of the hierarchical index (i.e., the I.S structure) on top of the set of inverted lists in each partition. The algorithm is invoked in Algorithm 1 (i.e., after the creation of a new partition in the first SSF level) and in Algorithm 2 (i.e., after the creation of a new partition by merging the partitions of an SSF level). The I.S structure is constructed sequentially and requires a single full scan of I.L previously created. The I.S tree is built from the leaves to the root. This requires one RAM page to scan I.L (line 1 in Algorithm 4), plus one RAM page to write the I.S (line 2 in Algorithm 4). For each Flash page of the I.L containing at least one head-list (line 8 in Algorithm 4), the maximum term and its Flash address are indexed in the index leaves (lines 10 and 13 in Algorithm 4). Once the bottom index level is created, the upper levels of I.S are trivially filled sequentially in the same manner through a recursive call (line 24 in Algorithm 4).

Algorithm 4. Build hierarchical index

Input: *ptr* the beginning address of the sorted inverted lists in a partition.
Output: *root* the address of the root Flash sector of the index.

1. ram_{in} = Alloc_RAM (sizeof(Flash_sector)) ;
2. ram_{out} = Alloc_RAM (sizeof(Flash_sector)) ;
3. ptr_{in} = *ptr* ; /* *pointer to the head sector of the list* $\{<t, f_t, \{(d, f_{d,t})\}_{\downarrow d}>\}_{\downarrow t}$ */
4. ptr_{out} = address of the next free sector in Flash ;
5. *no_nodes* = 0; /* *number of nodes in the currently built index level* */
6. **while** $ptr_{in} \neq \varnothing$ **do**
7. load in ram_{in} the Flash sector at address ptr_{in} ;
8. get max(*t*) in ram_{in} ; /* *find the last vocabulary term in this page* */
9. **if** $t \neq \varnothing$ **then**
10. append <*t*, ptr_{in}> to ram_{out} ;
11. **end**
12. **if** ram_{out} is full **then**
13. write ram_{out} at address ptr_{out} ;
14. *no_nodes*++ ;
15. **if** *no_nodes* == 1 **then**
16. *ptr* = ptr_{out} ; /* *keep the address of the first index node in the current index level for recursive call* */
17. **end**
18. ptr_{out} = ptr_{out} + sizeof(Flash_sector); /* *address of the next free sector in Flash* */
19. **end**
20. ptr_{in} = ptr_{in} + sizeof(Flash_sector); /* *get the address of the next Flash sector* */
21. **end**
22. free (ram_{in}); free (ram_{out});
23. **if** *no_nodes* > 1 /* *if the current index level has more than one node then recursively index this level* */
24. *ptr* = Build_hierachical_index(*ptr*);
25. **end**
26. **return** *ptr* ;

Algorithm 5 presents the computation of the F_t values for the query terms, which represents the first phase of the query processing. F_t is computed only once for each term *t* since F_t is constant for *Q*. This is why F_t is materialized in the dictionary part of the index ($\{t, F_t\} \subset I.S$), as shown in Figure 4. Since *I* is split in $<I_0, I_1, \ldots, I_p>$, the global value of F_t is computed as the sum of the local F_t of all partitions (lines 2 to 10 in

Algorithm 5). The algorithm visits all the SSF partitions (lines 2 and 3 in Algorithm 5) and in each partition it uses the *I.S* structure to access the inverted lists corresponding to the query terms (lines 4 and 5 in Algorithm 5). If a query term is found, its global F_t value is increased with the local f_t value (line 10 in Algorithm 5). For the sake of simplicity, we do not consider in Algorithm 5 the case of the documents overlapping between consecutive partitions. The overlapping documents are detected by checking the two bits (i.e., *firstd* and *lastd*) in *I.S* (see Figure 11). Whether an intersection between two lists is detected, the sum of their respective F_t must be decremented by *1*. Hence, the correct global value of F_t can easily be computed without physically accessing the inverted lists. During the F_t computation phase, the dictionary of each partition is read only once and the RAM consumption sums up to one buffer to read each dictionary, page by page, and one RAM variable to store the current value of each F_t. In addition, a set of pointers to the start Flash addresses of the inverted list for each query term in each SSF partition is also stored in RAM to avoid re-accessing the *I.S* of each partition in the second phase of the query processing.

Algorithm 5. Compute *Ft*

Input: a set $Q = \{q_i\}$ of q query terms, $Ft[q]$ an array of Ft scores with $Ft[i]$ the score for q_i.
Output: $Ptr[q][l][b]$ a set of pointers with $Ptr[q_i][l_j][p]$ having the start Flash address of the inverted list for query term q_i of partition p in level l_j.

```
1.  ptr a pointer to store the address of a Flash sector;
2.  for l = 0 to max ({ i | P[i][0] ≠ ∅}) do      /* for each level starting from the first one */
3.    for p = 1 to max ({ j | P[l][j] ≠ ∅) do     /* for each partition of that level */
4.      for i = 1 to q do                          /* for each query term */
5.        ptr = Access_hierarchical_index(P[l][p], qi); /* find the Flash sector containing the
      inverted list of qi in this partition */
6.          load in RAM the Flash sector at address ptr;
7.          search in RAM the entry of the element e = {qi, ft, {(d, fd,t)}↓d}↓t ;
8.          if e exists then
9.            Ft[i] = Ft[i] + e.ft;
10.           compute Ptr[i-1][l][p-1] as the address in Flash of the start of the list e.{(d,
      fd,t)}↓d ;
11.         end
12.     end
13.   end
14. end
15. return Ptr;
```

Algorithm 6 presents the pseudocode to compute the top-k document identifiers and their scores for a set of query terms, which represents the second phase of the query processing in the SSF. The algorithm takes as input the F_t values of the query terms and set of pointers to the start Flash addresses of the inverted list for each query term in each SSF partition, priorly computed by Algorithm 5. The proposed algorithm works as follows. For each SSF level from the lowest to the highest one (line 5 in Algorithm 6), all the partitions of the SSF level are accessed from the most recent to the oldest one (line 6 in Algorithm 6). For each partition, the *tf-idf* computation sums up to a simple linear pipeline merging process of the inverted lists for all terms $t \in Q$ (lines 8 to 32 in Algorithm 5). The RAM consumption (line 1 in Algorithm 6) for this phase is therefore restricted to one buffer (i.e., a RAM page) per query term t to read the corresponding inverted lists $I_i.L_t$ (i.e., $I_i.L_t$ are read in parallel for all t, the inverted lists for the same t being read in sequence). In addition, two lists are maintained in RAM (lines 2 and 3 in Algorithm 6) : *R[k] = {(d, score(d))}* contains the current k best *tf-idf* scores of documents which exist with certainty (no deletion has been encountered for these documents); *Ghost = {(d, score(d))}* contains the list of documents which have been deleted (a pair $(d, -f_{d,t})$ has been encountered while scanning the inverted lists) and have a score better than the smallest score in *R[k]*. The *tf-idf* score of each document d is computed (line 15 in Algorithm 6) by considering the modulus of the frequencies values $|\pm f_{d,t}|$ in the *tf-idf* score computation, regardless of whether d is a deleted document or not. *R[k]* and *Ghost* lists are managed as follows. If the score of the current document d is worse than the smallest score in *R[k]*, it is simply discarded and the next document is considered (line 16 in Algorithm 6). Otherwise, two cases must be distinguished. If d is a deleted document (a pair $(d, -f_{d,t})$ is encountered), then it enters the *Ghost* list (line 18 in Algorithm 6); else it enters the *R[k]* list unless its *id* is already present in the *Ghost* list (lines 20 to 22 in Algorithm 6). Note that this latter case may occur only if the *id* of d is smaller than the largest *id* in *Ghost*, making the search in *Ghost* useless in many cases. An important remark is that the *Ghost* list has to register only the deleted documents which may compete with the k best documents (line 23 in Algorithm 6), to filter them out when these documents are later encountered, which makes this list very small in practice. In addition, the probability that the score of a *Ghost* list element competes with the *R[k]* ones decreases over time, giving the opportunity to continuously purge the *Ghost* list (line 23 in Algorithm 6). If the *Ghost* list overflows (i.e., exceeds the size of a RAM page), it is sorted in descending order of the document ids, and the entries corresponding to low document ids are flushed.

For simplicity, we omitted the flushing process from Algorithm 6. Note also that this situation highly improbable as explained in Section 6.3. None of the queries we evaluated in our experiments has produced an overflow of the RAM page allocated to the *Ghost* list.

Algorithm 6. Compute Top-*k*

Input: $Q = \{q_i\}$ a set of q query terms ; $R_u=\{To_i\}$; $Ft[q]$ an array of size q with $Ft[i]$ the Ft values of the query term q_i ; $Ptr[q][l][b]$ a set of pointers with $Ptr[q_i][l_j][p]$ storing the address in Flash of the inverted list element $<t, f_t, \{(d, f_{d,t})\}_{\downarrow d}>$ for partition p, level l, and a term q_i ; k the number of documents identifiers requested in result.

Output: $R[k]$ an array of couples (d, *tfidf_score*) of document identifiers and their *tf-idf* score.

1. *Ram*[ ] an array of q sectors allocated in RAM ;
2. $\forall x \in [0, k\text{-}1]$, $R[x].tfidf\_score = 0$; /* *initialize the tfidf scores in the result at 0* */
3. *Ghost* one RAM page to temporarily maintain the current best ranked deleted documents still appearing in the index
4. /* *Multi-way merge of the inverted lists of the query terms in each index partition** /
5. **for** $l = 0$ **to** max ($l \mid \exists\, Ptr[q_i][l_j][p] \in Ptr, p \in [1, b], i \in [1, q]$) **do** /* *for each level containing query terms* */
6. **for** p = max ($p \mid \exists\, Ptr[q_i][l_j][p] \in Ptr$, $i \in [1, q]$) **to** 1 **do** /* *for each partition containing query terms* */
7. **for** each $i \mid \exists\, Ptr[q_i][l_j][p] \in Ptr$ **do** /* *for each query term* */
8. load in $Ram[i]$ the first sector of the inverted list $\{(d, f_{d,t})\}_{\downarrow d}$ of q_i from address $Ptr[i][l][p\text{-}1]$;
9. $Ptr[i][l][p\text{-}1] = Ptr[i][l][p\text{-}1]$ + sizeof(Flash_sector); /* *compute the next sector address of the current inverted list of* q_i */
10. **end**
11. $d_{frontier}$ = min ({$\forall i \in [1, q]$, max ({$d \in Ram[i]$}) }); /* *find the border document id for all the documents in RAM, i.e., all the doc ids inferior to the border doc id can be safely scored* */
12. **while** ($\exists$ a couple $(d, f_{d,t}) \in Ram \mid d \leq d_{frontier}$) **do** /* *compute tfidf for the next docs* */
13. d_{next} = min ({$d \in Ram$});
14. E = {couples $(d, f_{d,t}) \in Ram \mid d = d_{next}$};
15. *tfidf_score* = Compute_TFIDF(d_{next}, E, F_t); /* *compute TF-IDF for* d_{next} */
16. **if** *tfidf_score* > min ({*tfidf_score* $\in R$}) **and** *(Allowaccess(R_u, d, Read)==* ***TRUE)*** **then** /* *if the score of* d_{next} *enters the current best k scores* */
17. **if** d_{next} is a deleted document **then**
18. insert (d_{next}, *tfidf_score*) into *Ghost*
19. **else**
20. **if** $d_{next} \notin$ *Ghost* **then**
21. delete from R the entry with minimum *tfidf_score* ;

```
22.                 insert in R (d_next, tfidf_score) ;
23.                 delete from Ghost all entries d_ghost having score(d_ghost) < min_score(R) ;
24.                 remove from Ram all the couples e ∈ E;
25.               end
26.             end
27.           end
28.           for each empty sector Ram[i] ∈ Ram | Ptr[i][l][p-1] ≠ ∅ do /* scan next sectors */
29.             load in Ram[i] the inverted list {(d, f_d,t)}↓d of q_i from address Ptr[i][l][p-1];
30.             Ptr[i][l][p-1] = Ptr[i][l][p-1] + sizeof(Flash_sector);
31.           end
32.           d_frontier = min ({∀i ∈ [1, q], Max ({d ∈ Ram[i]}) }) ;
33.       end
34.     end
35. end
36. free(Ram); free(Ptr);
37. return R;
```

4. Prototype Platform

In this section, we present our prototype platform and describe the demonstration scenario covering the security and the performance of the proposed solution for the Personal Cloud paradigm.

4.1 Demonstration Platform

The Hardware Platform. The demonstration platform is an instance of the architecture presented in Figure 1. A PC plays the role of the home cloud data system organizing the document dataspace. In addition, the secure co-server manages the encryption/decryption of documents, indexes them, processes the queries and enforces the defined access control rules. The secure co-server runs on a tamper-resistant token produced by the ZED company and has an architecture representative for secure tokens (see Figure 17). Our secure token has the following characteristics: the MCU is equipped with a 32 bit RISC CPU clocked at 120 MHz, a crypto-coprocessor implementing AES and SHA in hardware, 128 KB of static RAM and 1 MB of NOR Flash. The MCU is connected to a smartcard chip hosting the cryptographic material and to a µSD card reader where the inverted index resides. The documents themselves are stored encrypted on the PC (i.e., the home cloud data system).

The secure token can communicate with the outside world either though the USB or the Bluetooth protocol. It is also equipped with a fingerprint reader providing a strong physical authentication of the queriers. The fingerprint reader can be used to identify the owner of the token that can either connect to her own token with all privileges or connect to another user's remote token with restricted privileges, which are regulated by access control rules established by the owner of the remote token.

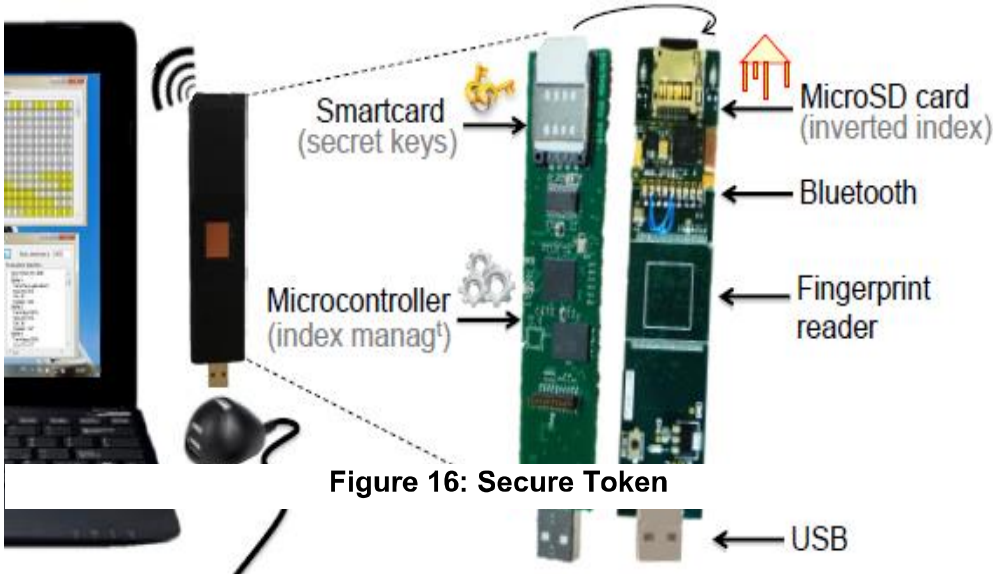


Figure 16: Secure Token

The Graphical User Interface (GUI). The demonstration interface reflects the external APIs of our secure search engine enabling the insertion and deletion of sets of terms (or tags) extracted from sets of documents, the keyword search and the definition of access control rules for the different subjects (see Figure 18, left part). When a given subject (e.g., an application, the data owner or a remote user) opens a session with the secure token, it first chooses an inverted index corresponding to a particular dataset (for more information about the datasets used in the experiments see Chapter VI). Then, it can add (or select) a set of files to (from) a (virtual) directory, to be inserted into (or deleted from) the index. Terms are extracted from these files and are completed with the appropriate tags before being inserted in the index embedded in the secure token. After insertion, the files appear encrypted on the host. The encryption key is automatically generated by the token from the document identifier and the secret key stored in the SIM smartcard and is unique for each file. Then, the encryption can be either performed by the host or by the token.

The connected subject is also able to issue keyword search queries to retrieve the relevant files. The relevant documents are ranked according to the *tf-idf score* and the *top-k* results are returned to the user. A link provides access to each result file, which can be decrypted on demand. The search queries conform to the access control rules. However, in the demonstration interface, a rule can be dynamically edited to show the effect on the result and the efficiency of the system whatever the access rule.

To demonstrate the behavior of the index, the interface also plots (see Figure 17, right upper part) the logical view of the index partitions and their corresponding physical view of the blocks/sectors in the Flash memory. Many useful statistics that describe the current state of each index partition, as well as the cumulative values maintained since the index creation, are also provided. For each insertion, deletion and search query, a synopsis of the underlying evaluation algorithm is also pictured (see Figure 17, right lower part) along with detailed statistics of their respective RAM consumption and IO patterns.

4.2 Demonstration Results

Security. The primary result is building a secure personal cloud platform where the evaluation of the access control rules is enforced by a secure token is within reach. The trustworthiness in the platform is rooted in two things. First, compared to a solution centralizing the personal data of millions of people in a single proprietary Cloud, the intrinsic decentralization of the Personal cloud decreases the incentive for attacks. Second, the use of secure hardware at the user's side strongly increases the difficulty and cost of an attack. Indeed, (1) the secure token provides a closed execution environment (no code can be installed and executed by the MCU but the original code and cryptographically signed versions of its upgrades), (2) the attacker must be in physical possession of the token to attack it, (3) tamper-resistance provides state of the art guarantees against physical and side-channel attacks, (4) the data at rest (stored in the NAND Flash) are cryptographically protected, (5) strong authentication is mandatory to unlock the personal server and no DBA or any other third party interfere in the loop. Hence, the ratio cost/benefit of an attack, which provides an accurate measure of the security level of a system, is greatly improved.

Bounded RAM. The second result is to demonstrate that a full-fledged search engine integrating access control can manage hundreds of thousands of documents while consuming kilobytes of RAM. The RAM consumption is indeed bounded by the number

of terms involved in the query and in the access control rules and remains independent of the size of the document set. This results from the ability to execute any query in a pure pipeline way even in the case where the document set is subject to updates (modifications and deletions of documents). see algorithms related to queries evaluations and documents insertions in subsection 3.

Full scalability. The third result is to show that improving query execution time is not always synonym of insertion/update degradation and vice-versa. This results from a sequential partitioned indexing structure providing linear performance for queries and insertions for a limited number of documents combined with a background reorganization process reaching logarithmic performance while the document set increases in size. The demonstration shows that the storage penalty incurred by this strategy is rather acceptable.

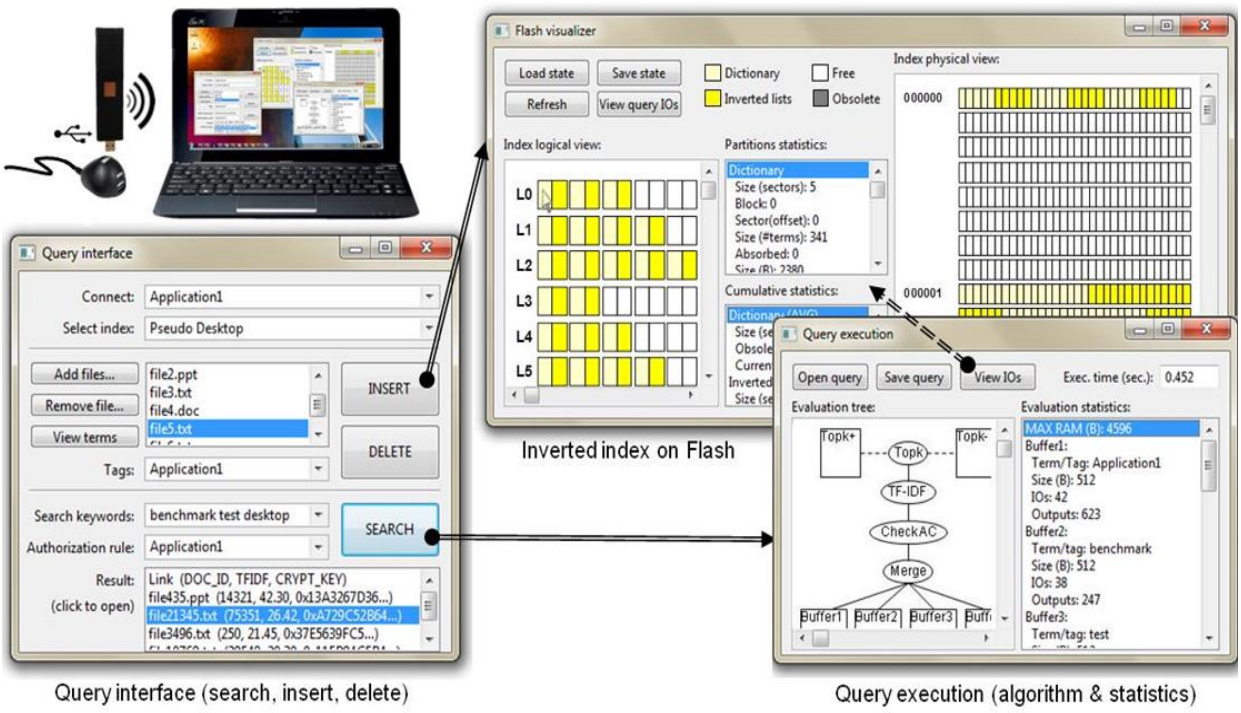


Figure 17: Demonstration Graphical User Interface

5. Conclusion

This chapter focuses on the implementation issues of the proposed search engine. We present first an overview of the KISS project and its relation with the Personal cloud, and explain our contributions in this project. Then, we present in detail the algorithms used to implement all the index operations and show that by construction, these

algorithms respect the RAM Bound requirement. Finally, we describe the prototype platform implementation using a GUI to show the internal structure of the index, the scalability, the RAM Bound and the security of the proposed solution.

Chapter VI
Performance Evaluation

In this Chapter we present an extensive evaluation of the proposed search and the access control engine. We introduce the testing hardware platform, the used datasets and the related use-cases in Section 1. In Section 2, we discuss the index maintenance, i.e., insertion/merge cost and the frequency of the merges. Our search engine has the double functionality as a search engine and an access control system. Hence, we compare the performance of our search engine with the state of the art techniques with and without access control verifications. The query performance of the search engine is presented in Section 3 .The impact of the deletion rate on the index size and the query performance is discussed in Section 4. Then, we discuss the impact of the access control in Section 5. We compare both the search and the insert performance of our method with two representative state of the art search engines techniques in Section 6. Finally, in the light of the obtained experimental results, we discuss the limitations of our approach in Section 7 and conclude this chapter.

1.Experimental Setup

Unit Test. All the experiments have been conducted on a development board ST3221G-EVAL (see www.st.com/web/en/catalog/tools/PF251702) equipped with the MCU STM32F217IG (see www.st.com/web/catalog/mmc/FM141/SC1169/SS1575/LN9/PF250172) connected to a MicroSD card slot (see Figure19, above part) and have been verified on a real smart token (see Figure 19, below part). The development board hardware configuration is representative of many typical smart objects [5] [63] [2] [51]. It runs the embedded operating system RTOS 7.0.1 provided in open source (see freertos.svn.sourceforge.net/viewvc/freertos/tags/V7.1.0/). The search engine code is stored in the internal NOR Flash memory of the MCU, while the inverted index is stored on a MicroSD NAND Flash card. We use for data storage two commercial MicroSD cards (i.e., Kingston MicroSDHC Class 10 4GB and Silicon Power SDHC Class 10 4GB) which exhibit different performances (the performance spectrum of microSD card performances is large, the performance of the chosen cards for the experiments are shown in lines 1 and 4 of Table 1 in Chapter II). The MCU has 128KB of available RAM. However, we only allocate a maximum amount of 5KB of RAM to the search

engine to validate our design for highly constrained existing smart objects like sensors. The development board includes a COM port that helps us trace the experiments behavior, unlike the real secure token which is less easily unobservable.

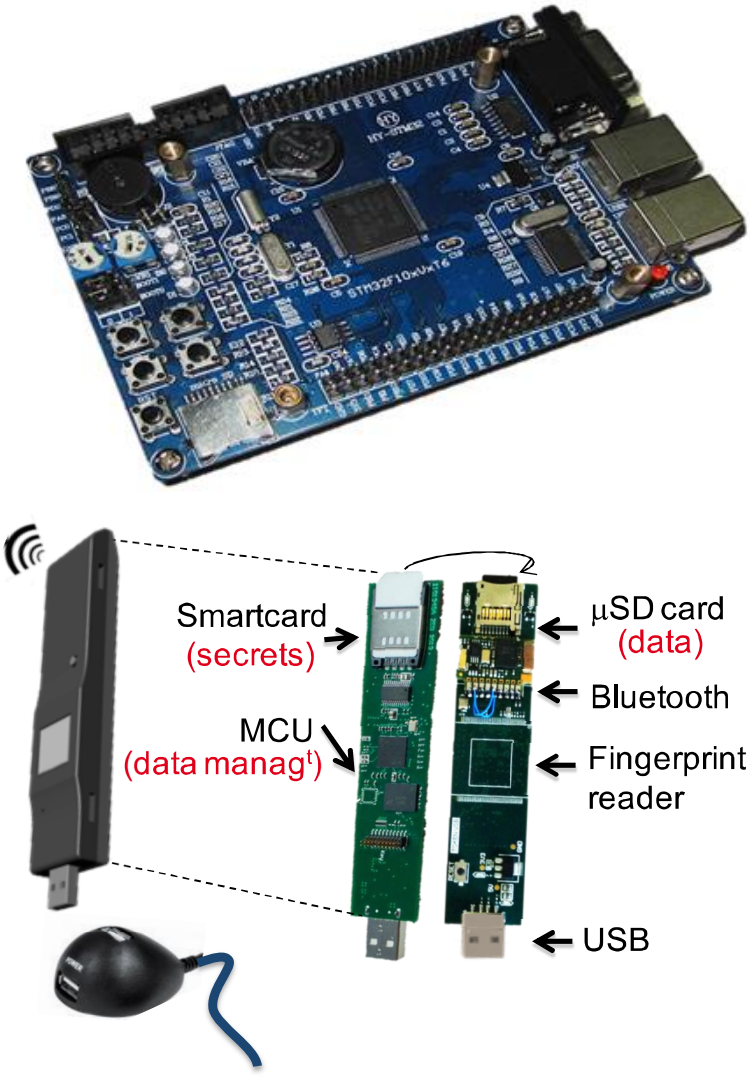


Figure 18: The development board ST3221G-EVAL used in the experiments (above) and the real hardware plateform (below).

Datasets and queries. Our evaluation is based on two datasets (see Table 2), which, by their diversity, cover the Personal Cloud use-cases. To evaluate the scalability and efficiency of the search engine, we use the ENRON dataset (available at https://www.cs.cmu.edu/ ~enron/) composed of 0.5 million emails and a query set of 300 representative queries prepared for this dataset (available at

http://www.prism.uvsq.fr/~isap/files/ENRON_queries.zip) built for this dataset. The interest of using this dataset is that it is large enough to test the index scalability and is well recognized in the IR community. The statistics of the dataset and the queries are given in Table 2. The total number of terms extracted from the ENRON dataset is 565,343 terms, among which 30.624 are frequent. We did not do any text pre-processing (e.g., stemming, feature selection, stop word removal, correction of typos, etc.) of the dataset, which partially explains the very large number of unique terms. However, the text processing is orthogonal to this work since our main concern is the search engine efficiency and not its accuracy. Also, the high number of terms is useful to show the scalability of the proposed search engine.

The second dataset is represented by the pseudo-desktop collection of documents presented in [65]. The prominent difference from ENRON is that this dataset contains five representative types of personal files (i.e., email, html, pdf, doc and ppt) and not only emails as in ENRON. The desktop search is an important topic in the IR community. However, real personal collections of desktop files cannot be published because this raises evident privacy issues. Instead, the authors in [65] propose a method to generate synthetic (pseudo) desktop collections and show that such collections have the same properties as real desktop collections. We use in our experiments the pseudo-desktop collection provided in [65]. The statistics of this collection are given in Table 2. As recommended in [65], we preprocess the files in this collection by removing the stop words and stemming the remaining terms using the Krovetz stemmer. This explains the smaller number of terms in the vocabulary (i.e., 337952) compared to ENRON (565343). Nevertheless, the vocabulary is still rich and contains a high number of terms. In addition, the average size of the files is about 8 times larger in the pseudo-desktop collection than in ENRON since the desktop collection contains not only emails but also larger html, pdf, doc and ppt files. In our experiments, we use a set of 837 queries prepared for this dataset and provided in [65].

Table 2: Statistics of the datasets and the query sets

Data set and query set	ENRON	Pseudo-desktop

Number of documents	500000	27000
Total Raw Text	946 MB	252 MB
Total Unique Words	565343	337952
Total Word Occurrences	52410653	35624875
Average Occurrences per Word	92	26
Frequent Words	30624	20752
Infrequent Words	534719	317210
Frequent Word Occurrences	5.41%	6.14%
Infrequent Word Occurrences	94.58%	93.85%
Size of documents in bytes (avg, max)	1KB, 874KB	8KB, 647KB
Size of documents in words (avg, max)	180, 108026	1304, 105162
Total number of queries	300	837
Number of queries with 1, 2, and 3 terms	51, 179, 48	85, 255, 272
Number of queries with 4 and 5 query terms	22	172, 82

2.Index Maintenance

According to the algorithms presented earlier, the insertions and deletions of documents produce a sequence of index partitions which are subsequently merged in the *SSF*. Given the RAM_Bound of 5KB, we set in all the experiments the branching factor *b* of the intermediate levels in the *SSF* to 8, to decrease the merge frequency, and the branching factor *b'* of the last level in the *SSF* to 3, to absorb faster the document deletions since the partitions of the last level are the largest.

The insertion or deletion of a new document is very efficient, since the document metadata is preliminary inserted in RAM. Also, given the small size of the RAM_Bound, flushing the RAM content into the level L_0 of the *SSF* is fast; it takes on average around 6ms to write a partition in L_0 in all our experiments (see Table 3 and Table 4). Given the low cost of the metadata insertion and the marginal impact of introducing the access control to the document insertion/deletion, a few terms are added while inserting/deleting a document. We focus next on the *SSF* merge cost, which is periodically triggered (i.e., each time the number of flushed partitions in L_0 reaches the branching factor *b*). The merge of the partitions in L_0 generates a new partition in L_1, which may generate a subsequent merge from L_1 to L_2 and in subsequent levels.

Table 3 and Table 4 present the number of IOs for the flush and merge operations performed in the different *SSF* levels, and their execution times for the two datasets

on the two tested SD cards, after the complete insertion of all the documents in the datasets and the random deletion of 10% of documents. In our experiments, the deletions are uniformly distributed over the inserted documents and uniformly interleaved with the insertions. All these operations lead to an *SSF* with seven levels for the ENRON dataset (see Table 3), with 6 levels for the desktop dataset (see Table 4). The number of levels of the index structure grows with the number of inserted documents and the average size of a document. As expected, the merge time grows exponentially from L_0 to L_6, since the size of the partitions also increases by (nearly) a factor of b. It requires a few seconds to merge the partitions in the levels L_0 to L_3 and up to several minutes in L_4 to L_6. The merge time is basically linear with the size of the merged partitions in the number of reads and writes. The merge time can vary especially in the first three levels of the *SSF*, depending on the distribution of the terms in the indexed documents. However, the partitions begin to contain most of the term dictionary in L_3 and the variation of the merge time in the upper levels is less significant. Note that with the pseudo-desktop collection, only fragments of documents are inserted in the first SSF level since the documents are large. Nevertheless, the documents fragments are united in the subsequent SSF levels once the partitions are merged. This fragment condensation also explains the smaller partition sizes of the intermediate levels of the SSF with the desktop dataset compared with the other dataset.

Table 3: Statistics of the flush and merge operations with the ENRON dataset

	Flush [$RAM \rightarrow L_0$]	**Merge [$L_0 \rightarrow L_1$]**	**Merge [$L_1 \rightarrow L_2$]**	**Merge [$L_2 \rightarrow L_3$]**	**Merge [$L_3 \rightarrow L_4$]**	**Merge [$L_4 \rightarrow L_5$]**	**Merge [$L_5 \rightarrow L_6$]**
Number of Read IOs	1 (1)*	67 (72)	585 (738)	3510 (4210)	21034 (23229)	129818 (14550)	404578 (404578)

Number of Write IOs	9 (9)	82 (102)	463 (707)	2738 (3448)	17703 (19771)	119357 (137789)	389774 (389774)
Exec. time on Kingston (seconds)	0.008 (0.0084)	0.67 (0.82)	3.8 (5.6)	22.3 (29.7)	141.3 (158)	944 (1077)	3003 (3003)
Exec. time on Silicon Power (seconds)	0.0044 (0.0045)	0.41 (0.57)	2.54 (3.56)	15.2 (17.7)	92 (102)	604 (688)	1907 (1907)
Total number of occurrences	286265	35783	4473	559	70	9	1
No. of inserted docs between consecutive flushes/merges	2 (27)	17 (134)	132 (964)	1057 (4306)	8410 (20061)	61556 (106601)	150237 (150237)

* The numbers given in brackets are maximum values, other values are average values.

Table 4: Statistics of the flush and merge operations with the pseudo-desktop dataset

	Flush [$RAM \rightarrow L_0$]	**Merge** [$L_0 \rightarrow L_1$]	**Merge** [$L_1 \rightarrow L_2$]	**Merge** [$L_2 \rightarrow L_3$]	**Merge** [$L_3 \rightarrow L_4$]	**Merge** [$L_4 \rightarrow L_5$]
Number of Read IOs	1 (1)*	90 (92)	503 (617)	2027 (2570)	11010 (15211)	50997 (73026)
Number of Write IOs	9 (9)	71 (100)	339 (548)	1485 (2085)	9409 (14027)	47270 (66335)
Exec. time on Kingston (seconds)	0.008 (0.0084)	0.58 (0.77)	2.9 (4.44)	13.2 (19.1)	84.6 (124.4)	436 (615)
Exec. time on Silicon Power (seconds)	0.004 (0.0045)	0.38 (0.48)	1.94 (2.84)	8.67 (11.7)	54.5 (79.3)	278 (393)
Total number of occurrences	73277	9160	1145	143	18	2
No. of inserted docs between consecutive flushes/merges	0.42 (16)	3 (42)	24 (232)	189 (1193)	1453 (6496)	8906 (10547)

* The numbers given in brackets are maximum values, other values are average values.

Table 3 and Table 4 indicate that the more costly a merge is, the less frequent it is. Typically, the merges in L_5, which require many minutes to complete, occur after the insertion and deletion of about 61000 ENRON documents or 9000 desktop documents. Only 80 merges costing more than 20 seconds are triggered while inserting the complete set of documents and deleting 10% of the ENRON collection. Even if the costly merges are rare, blocking the index when a merge is triggered for a long duration

may be problematic for some applications. The merge operation is thus implemented in a non-blocking manner as explained in Section 5. After every RAM flush an additional time window of 340ms for Kingston SD card and 210ms for Silicon Power SD card is allocated to resume the current merge operation (if any). The time window is chosen as the minimum time limit (in practice, we increase the minimum time with 10% to avoid any risk of merge overlapping) to guarantee that the merge of a given *SSF* level will end before the next merge of the same level is triggered. After this time delay, the merge is interrupted and its execution is memorized again. In this way, the potentially high cost of a merge operation is spread among a certain number of flush operations.

Figure 19 compares the time to execute the merge operations in a blocking and non-blocking manner in the index levels from 3 to 6 with the ENRON dataset (similar results were obtained with the pseudo-desktop datasets). The merges in levels 0, 1 and 2 could not be represented because of their very low cost and high frequency. Also, we only represented the insertion of approximately half of the ENRON dataset in Figure 19, since this is sufficient to capture the overall index update performance, while allowing for reasonable image clarity. We can observe that the cost of the merge in L_6 of the *SSF* is 1907 seconds (32 minutes) with the SP storage, if the merge is performed in a blocking manner. However, this cost will be spread among the next 11483 insert/delete operations (each time the RAM is flushed) using non-blocking merges. Also, a merge triggered in a lower SSF level preempts the current merge in a higher level (if any).

Table 5 compares the maximum and the average insertion/deletion time in the index with the blocking and the background merge implementation. The time is measured as

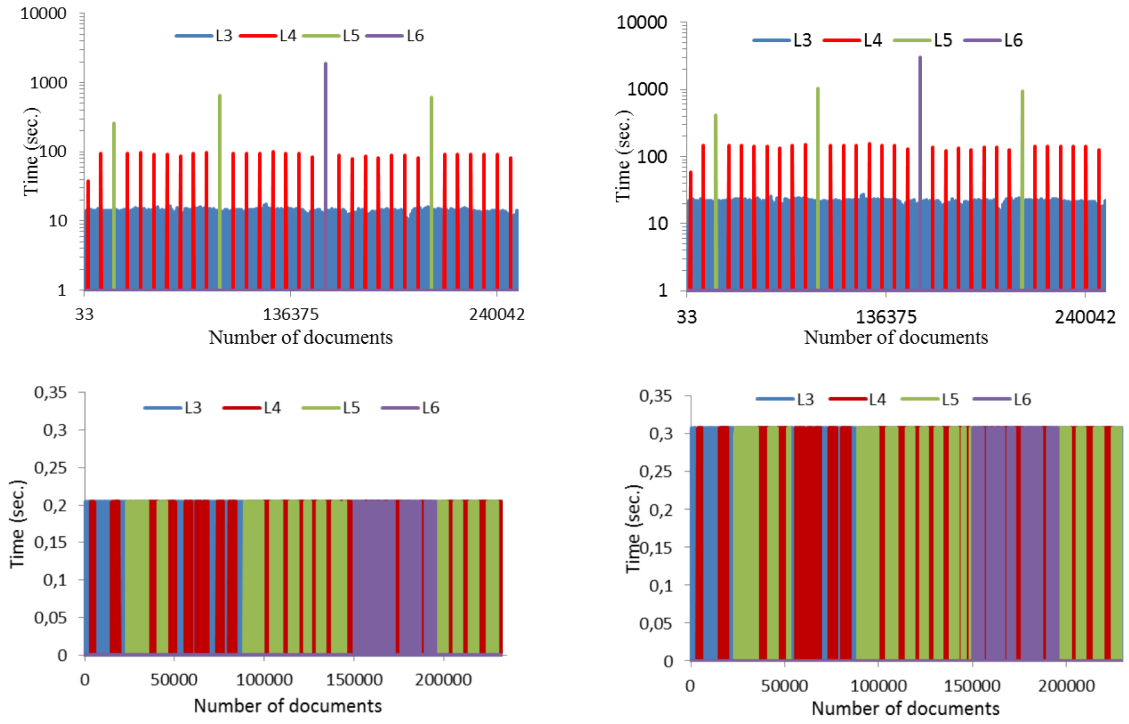


Figure 19: Insert performances with blocking (up) and non-blocking (down) merge with Silicon Power storage (left column) and Kingston storage (right column) for the ENRON dataset.

the RAM flush time plus the merge time, if a merge is triggered (for the blocking merge) or is currently in progress (for the non-blocking merge). With the blocking merge, there is a large gap between the maximum and the average insertion time, since the maximum insertion time corresponds to the merge time to create a partition in the sixth index level. With the background merge, this gap is much lower, since the cost of the merge operations is amortized over a large number of insertions/deletions. The insertion/deletion time never exceeds 0.31s for Kingston and 0.20s for Silicon Power (see Table 5). Note also that the non-blocking implementation of merges has a slight impact on the query cost since the number of lower level partitions can temporarily exceed the value of *b* (see next section).

Table 5: Blocking vs. non-blocking merge performance with ENRON

	SD Card	Max. cost (sec.)	Avg. cost (sec.)
Blocking merge	Kingston	3003	0.23

	SP	1907	0.15
Non-blocking merge	Kingston	0.31	0.29
	SP	0.20	0.18

3.Index Search Performance

We evaluated the search performance of the proposed index on our test board and the two SD cards, with both the blocking and non-blocking merge implementations. Because of the similarity of the results, we present in the figures hereafter the results obtained with both storages only with the ENRON dataset. For the pseudo-desktop datasets, we present the results with a single SD card, i.e., Silicon Power (SP). Figure 20 shows the average query time for the 300 search queries in our test query set, as a function of the number of indexed documents in the ENRON collection. The query set mixes queries consisting of 1 and up to 5 terms (see Table 2). For simplicity, the curves in Figure 20 present the query cost before (i.e., the "max" curves) and after (i.e., the "min" curves) each merge occurring in the higher index levels, i.e., from the level 3 to the level 5. We can observe that locally, the query cost increases linearly with the number of partitions in the lower levels, and then decreases significantly after every merge operation. The large variations in the query cost correspond to the creation of a new partition in the fifth level of the *SSF* (see the arrows in Figure 20), while the intermediary peaks correspond to the creation of a partition in level 4 of the *SSF*.

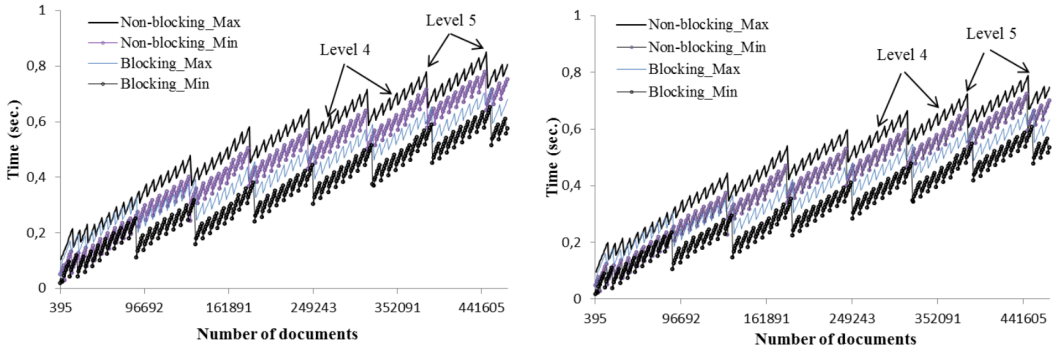


Figure 20: Query performances with blocking and non-blocking merge with Silicon Power (left) and Kingston (right) storage for the ENRON dataset.

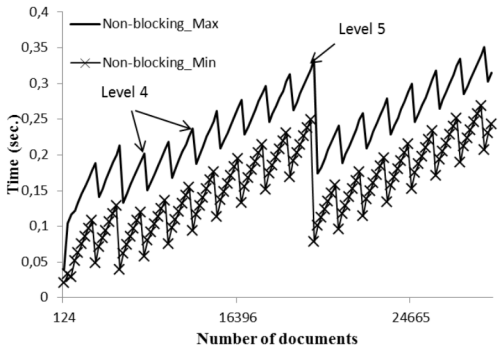


Figure 21: Query performances with blocking and non-blocking merge with Silicon Power storage for the Pseudo-desktop dataset.

Globally, the query time also increases linearly with the number of indexed documents, but the linear increase is very slow. For example, we see that after inserting 500K documents and deleting 10% of them, our search engine is able to answer queries with an average execution time of only 0.45s and a maximum time of 0.79s for Kingston and an average of 0.49s and a maximum of 0.89s for Silicon Power (with the non-blocking merge implementation). The query times are lower with a blocking merge, i.e., an average of 0.35s and a maximum time of 0.67s for Kingston and an average of 0.38s and a maximum of 0.72s for Silicon Power. The non-blocking merge leads to an increase of the query cost because the number of lower level partitions can temporarily exceed *b*, so that more partitions have to be visited. In our setting, the increase is on average of about 28% compared with the query time with a blocking merge. The query time increase is approximately 0.1s seconds for Kingston and 0.11 seconds for Silicon Power and appears to be a fair trade-off for applications that cannot accept unpredictable or unbounded update index latencies. Note that the increase of the query

cost with a non-blocking merge can be decreased by enlarging the time window allocated to periodically process a merge.

Figure 21 presents the evolution of the query time with the number of indexed documents for the pseudo-desktop. The deletion rate in this figure is 10% as with ENRON. For better readability of the figures, we present the minimum and the maximum query times only with the non-blocking merge implementation. As with the ENRON dataset, the query times with a blocking merge are about 25% lower than with the non-blocking merge. With the pseudo-desktop dataset, our index exhibits an average execution time of only 0.17s and a maximum time of 0.32s for Kingston and an average of 0.18s and a maximum of 0.35s for Silicon Power. As with the ENRON dataset, globally the query cost increases linearly with the index size. The overall index size is lower for the pseudo-desktop dataset compared with ENRON (see Section 5.4), which explains the smaller values of the average query times with these two datasets.

4. Index Performance with Various Deletion Rates

In this section we discuss the impact of the deletion rate on the index search performance and index size. Since the deletions are executed as insertions in our search engine, the performance of delete operations is the same as with for the insert performance. However, the deleted documents are temporarily stored in the index structure (i.e., until the deletions are absorbed during the index merges), which leads to an increase of the index size compared to the optimal index size (i.e., where deleted entries are directly purged). The increased index size can degrade the query performance. Nevertheless, the experimental results show that the query performance is only marginally impacted by deletions even for high deletion rates.

Table 6 shows the average query performance for different deletion rates with the Kingston storage (we obtained similar results with the SP storage). Here, we considered two cases. First, we inserted the whole dataset while deleting a number of documents corresponding to the deletion rate (first line in Table 6). In this case, the higher the deletion rate is, the lower the query time is since a good part of the deleted documents (app. 50%) will be purged from the index and decreases the query processing time. In the second case, the total number of active documents in the index is the same regardless the deletion rate (second line in Table 6). Hence, the higher the

deletion rate is, the more documents we insert to compensate the deletions. In this case, a higher deletion rate leads to larger query times since part of the deletions are present in the index and have to be processed by the queries. However, the increase of the query times is relatively small compared to the case with no deletions, i.e., less than 12% for deletion rates up to 50%. Globally, the index is robust with the number of deletions in both cases.

Table 7 shows the index size for the two datasets after the insertion of all the documents in the collection and the uniform deletion of a certain percentage of the indexed documents. In each table cell, the first number indicates the cumulated size in MB of all the *I.L* parts of the SSF (i.e., the global size of the inverted lists), while the second number gives the cumulated size of all the *I.S* parts of the SSF (i.e., the global size of the search structures). Also, we give in Table 7 for each dataset and deletion rate the index size of the classical inverted index used as reference. Without deletions, the SSF index size is practically the same with the inverted index size. The SSF requires a little bit more storage for the *I.S* since each partition has its own search structure to index the terms in the partition. Nevertheless, the storage overhead of the SSF is negligible since the search structure represents less than 1% of the global index size (i.e., the inverted lists require much more storage than the search structures). With deletions, the size of the SSF index is larger than the size of the inverted index. Also, the size difference increases with the delete ratio. The explanation is that the deleted documents are reinserted in the SSF, which temporarily increases the index size. However, when a merge is triggered in an index level, a part of the deletions are absorbed and the deleted documents are purged from the index. Therefore, at any given time only a part of the deleted documents are still present in the index. Typically, in our tests we observed that about 45% to 55% of deletions are not absorbed after a high number of document insertions and deletions. This makes the SSF index size to be at most 40% larger than the inverted index size even for high deletion rates, which is quite acceptable.

Table 6: Average query performance (in sec.) with different deletion rates and Kingston storage

	0%	10%	30%	50%

Avg. query time (ENRON: 500k docs)	0.49	0.45	0.41	0.37
Avg. query time (ENRON: 250k docs)	0.33	0.34	0.35	0.37
Avg. query time (Desktop: 27k docs)	0.18	0.17	0.16	0.15
Avg. query time (Desktop: 13k docs)	0.12	0.13	0.15	0.16

Table 7: Index (inverted lists/search structures) size (MB) with different deletion rates (from 0% to 50%)

	0%	**10%**	**30%**	**50%**
SSF I.L/I.S size (ENRON: 500k docs)	439/3.5	402/3	350/2.4	310 /2.2
Inverted index I.L/I.S size (ENRON: 500k docs)	439/0.6	397/0.6	305/0.6	220 /0.6
SSF I.L/I.S size (Desktop: 27k docs)	81/1.24	76/1.14	60/0.82	55 /0.73
Inverted index I.L/I.S size (Desktop: 27k docs)	81/0.4	73 /0.4	57 /0.4	40 /0.4

5. Index Search Performance with Access Control

In this section we present the impact of the access control rules on the query performance. We limit the scope of the evaluation to conjunctive access control rules for the sake of simplicity. To generate queries with access control rules we associate to each query a set of access terms, i.e. terms involved in the currently active (conjunctive) access control rule. To test the impact of access control on the search engine performance we vary first the selectivity of the involved access terms, and second the number of access terms involved in each access control rule. To be able to compare the results with our previous measures, we consider the same datasets as previously, namely the Enron data set and the Pseudo-desktop one, and the same Silicon Power microSD card.

In the next experiments, we have fixed the *Top-k threshold* to 10 (i.e., k=10).

Case 1 : Impact of the selectivity of the access terms on the query performance.
To test the impact of the selectivity of the access terms on the query performance, we select sets of terms appearing with different frequencies in the considered data sets. We build three groups of 100 terms. The first group contains 100 terms appearing in the documents collection with a low frequency (i.e, an average of 27), group 2 contains terms with a medium frequency (an average of 325) and group 3 contains terms with a high frequency (an average of 236525). We then randomly select 100 queries from

the query set, and we associate this query with one access term from each group. We measure the performance of each query without any active access rule, with an active rule made of a single access term taken from group 1 (the query is considered as belonging to the query group 1), taken from group 2 (the query belongs to query group 2) and taken from group 3 (query is in the query group 3).

Table 8 shows the average query performance of the queries according to their query group, after insertion of the whole data set. We can see that the impact of the access terms selectivity on the query performance is very low for groups 1 and 2, while it is higher for group 3. With the ENRON query set, the search engine is able to answer queries with an average execution time of 0,77s without AC rules, 0,82s for groups 1 and 2, and 1,34s for group 3. With the pseudo-desktop query set, the average execution time of queries without AC was 0,27s for query with groups1 and 2 was 0,29s and for group 3 was 0,43s. Globally, the query performance of groups 1 and 2 increases of about 8% compared to queries without AC rules and about 68% for the queries of group 3. As the statistics of the data sets in table 2 show, only 5% of the terms are frequent in the Enron data set and 6% in the Pseudo-desktop data set. The frequent terms are not the common case in a data set dictionary and generally the query performance with access control rule will be marginally impacted by adding the AC rule as the results of group 2 show.

Our experiments show that adding one frequent access term to the query can increase the execution time by 6s for the Enron data set (the largest data set we had). The good query performance for queries of group 3 is explained by its evaluation strategy. As demonstrated in section 6.3, there are just few documents entering the *Top-k* list during the query evaluation and just for those documents the AC is checked by reading the access terms inverted lists (possibly in dichotomy), which leads to avoid reading the whole list thus reducing significantly the IOs cost especially for frequent terms.

Table 8: Impact of access terms selectivity on the queries performances with silicon power storage (in seconds).

	Without AC	Group1	Group2	Group3
Enron	0,77	0,82	0,82	1,34
Pseudo-desktop	0,27	0,29	0,29	0,43

Case 2: Impact of the number of access terms on the query performance.

To meet the different needs of a personal cloud, access control rules may contain several access terms. To understand the overhead of the number of access terms on the query performance, we consider access rules which contain 3, 5 and 7 terms randomly chosen from the data set. We generate 100 queries for each data set.

Table 9 shows the average query time of the queries without access control, and in the presence of an access rule involving 3 access terms (group 1), 5 access terms (group 2) and 7 access terms (group 3). The queries are evaluated after inserting the complete respective data set. We can see that the search engine is able to answer queries with an average execution time of 1s for queries of group 1, 0.97s for queries of group 2 and 0.93s for queries of group 3 for the ENRON data set. A similar behavior is noticed with the Pseudo-desktop data sets, where the average execution time of queries is 0.35s for queries of group1, 0.34s for group 2 and 0.33s for group 3.

These experimental results show that the query performance is only marginally impacted by the number of tags in access control rule with a maximum increase of 30% compared to query without AC. The good performance of query execution of group 3 compared to query of group 2 and 1 can be explained by our evaluation strategy applied to conjunctive rules. Indeed, we evaluate the access control rules by starting with the least frequent access term, which avoids accessing the subsequent access terms if the condition on the current term is not satisfied. Thus, many inverted lists corresponding to the most frequent access terms do not need to be accessed.

Table 9: Impact of number of access terms in the AC rules on the query performance with silicon power storage (in seconds)

	Without AC	Group1	Group2	Group3
Enron	0,77	1	0.97	0.93
Pseudo-desktop	0,27	0.35	0.34	0.33

6. Comparison with the State-of-the-Art Search Engine Methods

The existing solutions face important limitations when used to index large collections of documents in secure token. Classical inverted index [24] designed for magnetic disks does not comply with the Flash storage constraints. Besides, new methods [2] have been proposed recently and consider the constraints of smart objects to process

the insertions efficiently, but they do not scale with large datasets and cannot support index updates.

In this section we compare our proposed search engine (called SSF) with the representative indexing method of each of the aforementioned approaches. Hence, we choose the typical inverted index to represent the query-optimized index family (see Figure 4) and Microsearch [2] for the insert-optimized index family. Note that the other embedded search engines presented in Section 2.3 rely on similar index structures with Microsearch. We used the same test conditions as above, i.e., by using a RAM_Bound equal to 5KB, for the two competing methods. The data insertions in the inverted index method are processed in a similar way as in our search engine. The insertions are first buffered in RAM until the RAM_Bound is reached. Then, the buffered updates are applied in batch to the inverted index structure. However, different from our approach, the inverted index structure is modified in-place, which requires costly random writes. Also, to be able to evaluate the queries under the RAM constraint with the inverted index, the inverted lists of this structure have to be maintained sorted on the document ids, which permits applying a linear pipeline query processing similar to the SSF. In the case of Microsearch [2], we used a hash function with 8 buckets, since this value leads to the most balanced query-insert performance given the 5KB of RAM. Besides, we only considered data insertions and queries in the tests below, since Microsearch does not support deletions.

Insertion performance. Figure 22 and Figure 23 show the average insertion time for the three methods (i.e., SSF, Microsearch and the Inverted Index) with the two datasets. Microsearch and SSF have similar insert performance. On average, for the ENRON dataset, a document insertion in Microsearch takes about 0.094s with Kingston and 0.059s with Silicon Power, and 0.14s with Kingston and 0.09s with Silicon Power in SSF. In comparison, the document insertion time in the Inverted Index is much larger because of the costly random writes in Flash memory. On average, an insertion requires 30s with Kingston and 7.6s with Silicon Power, which is nearly two orders of magnitude higher than with the embedded search engines, and clearly dismisses this method in the context of constrained smart objects like secure tokens. We obtained similar results with the pseudo-desktop dataset as presented in Figure 23. On average, with Silicon Power storage, a document insertion with Microsearch takes about 0.08s, 0.33s with SSF and 30s with Inverted Index. For all the methods,

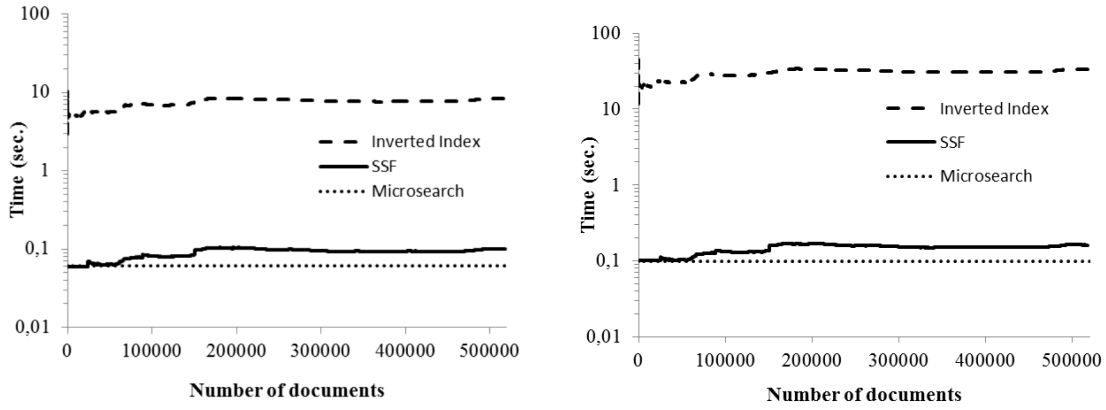


Figure 22: Average document insertion times with Microsearch, SSF and Inverted Index, with Silicon Power (left) and Kingston (right) storage, for the ENRON dataset

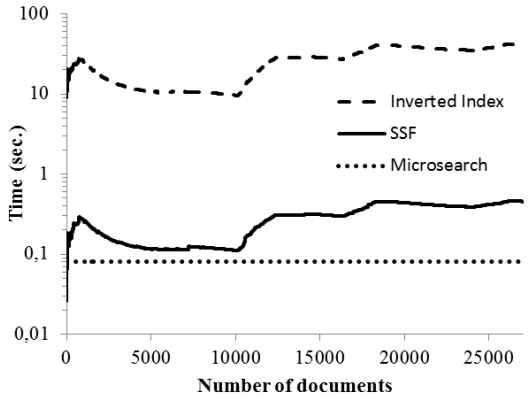


Figure 23: Average document insertion times with Microsearch, SSF and the Inverted Index, with Silicon Power storage for the pseudo-desktop dataset

the insertion times are proportional to the documents size. Therefore, the insertion times are the highest with the pseudo-desktop collection (larger documents).

The insertion with SSF and Microsearch relies on sequential writes in Flash to comply with the Flash memory constraints. The higher insertion cost of SSF compared to Microsearch is due to the SSF merges. Although the SSF insertions are less efficient than with Microsearch, the insertion time remains reasonable and will probably satisfy most of the applications especially if they require indexing large collections of documents. Indeed, the slight increase of the insert cost is outweighed by the query performance and scalability of the SSF.

Query performance. Figure 24 and Figure 25 show the query execution time for the three methods in function of the number of indexed documents. The Inverted Index has the best query times and can be considered as the most efficient structure for processing full-text search queries. We can see that query times with SSF are very close to Inverted Index times. On average, for the ENRON dataset, it takes 0.49s with Kingston and 0.53s with Silicon Power to process a query with SSF, while for with Inverted Index it takes 0.16s with Kingston and 0.17s with Silicon Power. Also, the average query times with Silicon Power is 0.18s with SSF, 0.07s with Inverted Index, and 880s with Microsearch, for the pseudo-desktop dataset. The difference in performance between the SSF and the Inverted Index methods is explained by the fragmentation of the index structure of the SSF. However, the index partitioning in the SSF is largely outweighed when we take into account the insert performance of these two index structures.

Microsearch has the worse query performance, which clearly is not scalable to a large number of documents. On average, for the ENRON dataset, to process a query with Microsearch, it takes 1728.80 seconds (28 minutes) with Kingston and 1861.78 seconds (31 minutes) with Silicon Power. Given these very large query times, we measure the real query time only up to 10000 indexed documents and estimate the query times above this number of documents, which is fairly simple since the query time linearly increases with the number of documents in Microsearch. Note also that even for a low number of documents, the query times with Microsearch are larger than the query times with SSF. For instance, for 1000 documents it takes 3.1s with Kingston using Microsearch and 0.1s using SSF. The first reason is that in Microsearch an inverted list corresponds to a large number of terms (i.e., all the terms are distributed in the 8 hash buckets). Therefore, a large part of the index data is retrieved to evaluate each query. Second, Microsearch requires two passes over the chained list containing a query term. The first pass is done to compute the global F_t value of the term, while the *tf-idf* score of the documents containing the term is computed in the second pass.

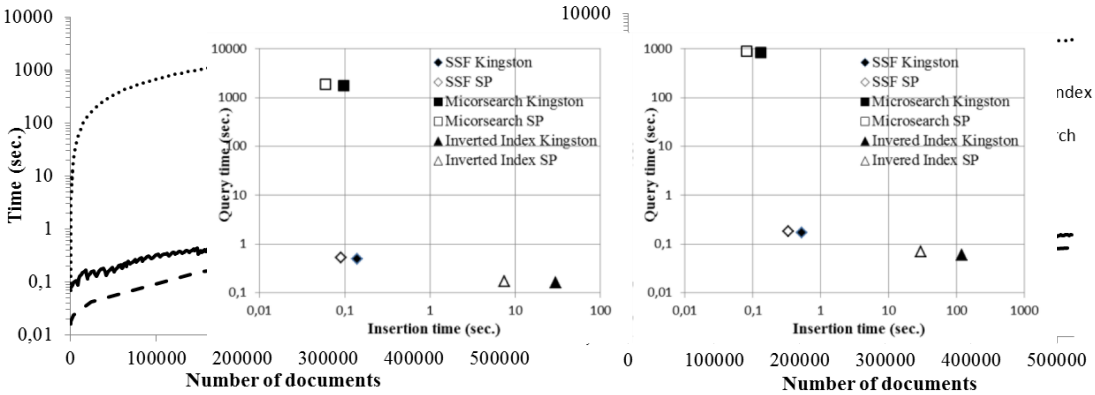


Figure 26: Overall performance comparison for the ENRON dataset (left) and for the pseudo-desktop dataset (right).

Figure 24: Query execution times with the Inverted Index, SSF and Microsearch, with Silicon Power (left) and Kingston (right) storage, on the ENRON dataset.

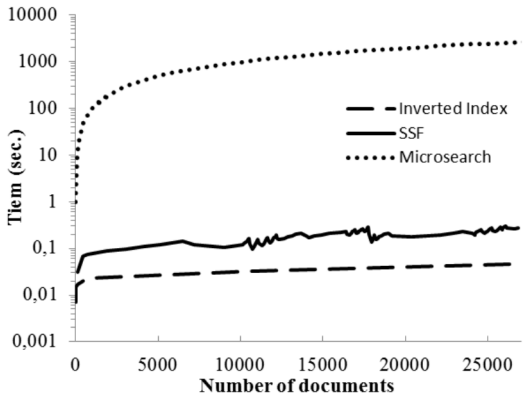


Figure 25: Query execution times with the Inverted Index, SSF and Microsearch, with Silicon Power storage, for the pseudo-desktop dataset.

Overall performance. The tiny RAM and the NAND Flash storage of smart objects introduce conflicting constraints on the index structure. Figure 26 summarizes the update and the query performance of the proposed index structure and the two representative competitors. Our solution is the only one to offer both query and update scalability under these constraints, by balancing the query and the update costs for any kind of document collection. At the same time, the loss in both the query and the update performances remains reasonable compared with the query optimized index (i.e., Inverted Index) and the insert optimized index (i.e., Microsearch).

7. Discussion

In the light of the above presented experimental results, we discuss in this section the limitations of our proposal. First, we analyze the limitations of the SSF in comparison with the classical inverted index, which can be considered as the ideal structure from the query performance and index size points of views. The shortcomings of the SSF originate from the major differences between the SSF and the Inverted Index structures, i.e., the index partitioning and the deletion processing.

The partitioning of the SSF influences both the query and the insert performances. The query performance is degraded because of the multiple searches in several small *I.S* (i.e., the search index dictionary of each partition), which are more costly than a single search in a large *I.S*. However, given the exponentially increasing size of the partitions in the SSF levels, the loss in the query performance is limited compared with Inverted Index (see Figure 24 and Figure 25). Also, the query performance loss is diminishing with the increase of the number of indexed documents, suggesting that our approach is particularly appropriated for indexing large datasets. We should also note that the partitioning introduces some variability in the query cost (see the stairway-like curves in Figure 20 and Figure 21). The partitioning has an impact on the insert performance since already inserted documents have to be rewritten at each index merge. Yet, the insertions only generate sequential writes in Flash and the insert cost is much more scalable than for the Inverted Index case (see Figure 23 and Figure 25). Finally, the partitioning has also an impact on the index size, since the search dictionary structures in each partition are redundant. However, the increase in the index size is negligible (see Table 7) as the search structures only take a small fraction from the global index size.

The specific way of processing deletions in the SSF also has an impact on the query performance and on the index size. Since a deleted document is first reinserted in the index, deletions lead to a temporal increase of the index size and consequently, of the query cost. Nevertheless, this negative effect is limited by the merge operations that permit to purge some of the deleted documents. The experimental results show that, even for high deletion rates of up to 50%, the increase in the index size is lower than 40%, while the increase in the query cost is lower than 12% (see Tables 6 and 7).

Of course, the size of the query result has also an impact. The query performance is indeed related to the number of documents entering the *Top-k* list, which is inversely proportional to the maximum size of the *top-k* list (i.e., the value of *k*). An alternative evaluation would be to execute the query in several steps, with a lower *k* value in each step. For example, the query evaluation with a *Top-k* list of 100 could be processed in 10 steps, each step producing 10^{th} relevant documents, from the 10 most relevant to the 10 less relevant of the result. This kind of evaluation would only require to store in each step the lowest score of the results produced in the previous step, and discard any document with a score above that value. Finally, another limitation of the proposed search engine is that we do not consider the problem of concurrent access, i.e., multiple processes that query/update the index at the same time. In our workloads combining queries/inserts/deletes, we analyze the cost of each separately, one operation at a time. Indeed smart objects, contrary to central servers, rarely support parallel or multi-task processing. Moreover, the RAM consumption increases linearly with the number of operation executed in parallel, a serious constraint in our context.

8. Conclusion

In this Chapter we discussed extensive experiments of the search engine using two datasets : Pseudo-Desktop which is an example of the documents found in a typical personal desktop, and ENRON which is mainly used to test the scalability of the index. We consider a hardware representative of smart objects, with a RAM_BOUND of 5KB, and we used two microSD cards with different performances for the storage of the index. Our measures show that the performance of the insertion is very efficient in the lower level of the index and a little costly in the higher levels. To solve this problem, we show that the merge operation can be implemented in background. For the query performance, it is also efficient and less than 1 second in average for most queries on both the datasets, even with the increase generated by the background merges.

The impact of the delete operation on the query performance and the index size is limited, since the deletions are executed as the insertions. The query performance is thus temporarily increased (before the deleted document can be removed during a subsequent merge operation). The increase is less than 12% even for high deletion rates (up to 50%). Regarding the index size, it is slightly increased by the deletes since they are temporarily sorted in the index subsequently. Deleted documents are absorbed and purged from the index during merge operations. Hence, a high amount

of insertion/deletion makes the index at most 40% larger than the typical size (which is quite acceptable) since these deletions are not yet absorbed.

We have shown that the impact of the access control on the query performance is acceptable, even in the case of highly frequent access terms and access rules involving many access terms.

We have also compared our search engine with two state-of-the-art methods, Microsearch as an example of the insert scalable family, and the typical Inverted Index as an example of the query scalable family. Our index has proven its superiority and full scalability compared to the state-of-the-art techniques and offers good insertion and query performances at the same time.

Finally, we have discussed the impact and limitations of our proposal on the index size, query performances and concurrent accesses.

Chapter VII
Conclusion and Perspectives

With the convergence of mobile communications, sensors and online social networks technologies, we are witnessing an exponential increase in the acquisition of personal data. For now, all these data end up in corporate servers or in the Cloud.

The Personal Cloud is a paradigm giving users the ability to store their complete digital environment and share it with other users and applications under their control.

It can be thought of as a dedicated box connected to the user's internet gateway, equipped with storage, computing and communication facilities, running a personal server and acquiring data from multiple source (such as smartphones, cameras, banks, web sites, employer, doctors).

To query such personal, data we proposed a simple keyword search engine, in the spirit of Google desktop or Spotlight. Terms are extracted from the files content and from metadata describing them (e.g., name, type, date, creator, tags set by the user herself). Since keywords seem the most convenient way to query the user dataspace, it make sense to let users express access control rules in a similar way, that is through logical expressions on terms associated to documents.

However, letting unexperienced users managing all these personal data exposes them to major security breaches. A common user is not a security expert. To make this architecture secure, we propose to employ a secure co-server to manage all the sensitive data (e.g., cryptographic keys, metadata, indexes) and data processing (e.g., query evaluation and access control). Hence, user personal data can be stored encrypted in a personal computer or the Cloud, but the metadata, the index to query this data and the encryption keys are stored in the secure co-server. In our case, this co-server plays the role of a secure Google desktop/Spotlight for the user's personal data. The high level of security and trust of this architecture is rooted in the combination of hardware and software security and data decentralization.

The proposed search and access control engine relies on three design principles, namely Write-once Partitioning, Linear Pipelining and Background Linear Merging. They are combined to produce an embedded search and access control engine reconciling high insert/delete rate and query scalability for very large datasets. The proposed solution was published in the VLDB conference [61], and its implementation

was demonstrated in the SIGMOD conference [63]. By satisfying a RAM_Bound agreement, our search engine can accommodate a wide population of smart objects, including those having only a few KBs of RAM. Satisfying this agreement is also a means to fulfill co-design perspectives, i.e., calibrating a new hardware platform with the hardware resources strictly necessary to meet a given performance requirement. The proposed search engine has been implemented on a hardware platform having a configuration representative of smart objects. The experimental evaluation validates its efficiency and scalability with real, large datasets and also demonstrates the superiority of this approach compared to state of the art methods.

The work conducted in this thesis can be pursued in various directions. We identify below some challenging issues and outline possible lines of thought to tackle them.

- **Generalization to other techniques of indexing.** Our three design principles can be generalized for building other kinds of embedded query engines (e.g., NoSQL like, B-tree, R-tree,…) considering that indexing any form of data streams in mass storage smart objects will encounter similar hardware constraints and then lead to similar requirements. For example, the traditional B-tree index used in DBMS systems differs from an inverted index only by the fact that the $f_{d,t}$ values are not present. Similar technologies may thus be applied. An additional difficulty will however be to maintain several indexes (B-tree) with little RAM.

- **Distributed Secure Search in the Personal Cloud.** In the high level of decentralization of the Personal Cloud, i.e., each user owns her personal cloud (see Figure 27), it makes sense to propose a distributed and secure search engine to the applications. This would be useful for any application developed for communities of users sharing a similar interest. For example, within a community of patients suffering from the same pathologies, each participating user may provide her own set of information such that, e.g., distributed searches may help to identify within the community the most relevant documents related to current treatments or symptoms. However, this must be performed without exposing the privacy of the participating users, which is not an easy task.

  First, to evaluate a global search, the scores computed in the different personal clouds must be comparable. This requires computing beforehand the global values of N and F_t to be used in the *tf-idf* formula. Then, the local top-k scores have to be exchanged and compared to find the global result. The secure

tokens, which constitute a source of trust in the architecture, can be used to perform the computation in a secure manner. However, in a context where the hardware is in users' the hands, the possibility of having a small percentage of hacked tokens (e.g., as a result of a sophisticated attack from the token owner) must be considered. This may lead to consider new metrics based on a risk analysis. Breaking a set of secure tokens should not put the personal documents of the whole community at risk. To evaluate that risk, new metrics may be based on (1) the amount of intermediate results and documents exposed during the evaluation of a distributed search query to potentially compromised tokens, and (2) the amount of information issued from a given honest participants which may be transmitted to the compromised tokens. The first metric gives an estimate of the benefit of an attack for the attackers and the second metric estimates the impact of the information leak for an honest participant. New distributed computation algorithms may be envisioned to minimize these metrics.

- **Challenge: Secure Sharing Model.** First, the AC model considered in this study exhibits a few limitations. Typically, the N and F_t values considered during the calculation do not take into consideration the real number of documents eligible for the access control (but consider the complete set of documents). As a future work, this problem needs to be corrected. In addition, different kinds of access control models may be supported by our search engine. As a future work, we can imagine supporting other kinds of access control models

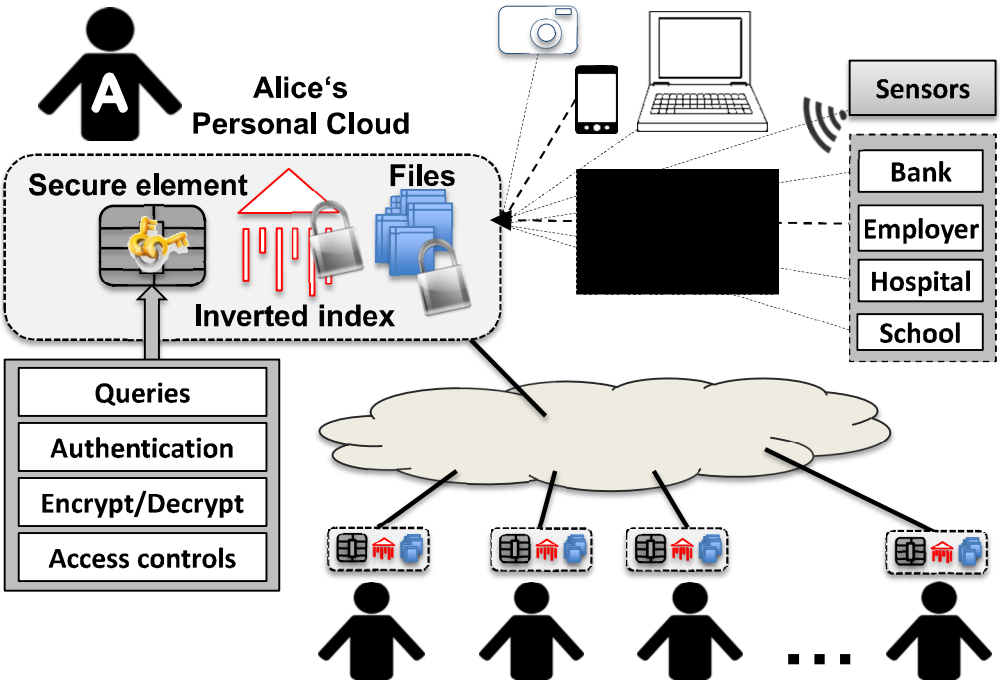


Figure 27 Distributed Secure Personal Cloud architecture

Bibliography

[1] C. C. Aggarwal, N. Ashish and A. Sheth, "The internet of things: A survey from the data-centric perspective," *Managing and mining sensor data. Springer US.,* pp. 383-428, 2013.

[2] C. C. Tan, B. Sheng, H. Wang and Q. Li, "Microsearch: When Search Engines Meet Small Devices," *Pervasive Computing,* vol. 5013, pp. 93-110, 2008.

[3] K.-K. Yap, V. Srinivasan and M. Motani, "MAX: Wide area human-centric search of the physical world," *ACM Transactions on Sensor Networks (TOSN),* vol. 4(4), August 2008.

[4] N. Anciaux, P. Bonnet, L. Bouganim , B. Nguyen, I. Sandu Popa and P. Pucheral, "Trusted cells: A sea change for personal data services," *CIDR,* 2013.

[5] N. Anciaux, L. Bouganim, P. Pucheral, Y. Guo, L. Le Folgoc and S. Yin, "MILo-DB: a personal, secure and portable database machine," *Distributed and Parallel Databases,* vol. 32(1), pp. 37-63, March 2014.

[6] C. C. TAN, B. SHENG, H. WANG and Q. LI, "Microsearch: A Search Engine for Embedded Devices Used in Pervasive Computing," *ACM Transactions on Embedded Computing Systems (TECS),* vol. 9 Issue 4, 2010.

[7] H. Wang, C. C. Tan and Q. Li, "Snoogle: A Search Engine for Pervasive Environments," *IEEE Computer Society,* vol. 21(8), pp. 1188 - 1202, Aug. 2010.

[8] Y.-M. Huang and Y.-X. Lai, "Distributed energy management system within residential sensor-based heterogeneous network structure," *Wireless Sensor Networks and Ecological Monitoring,* vol. 3, pp. 35-60, 2013.

[9] "Eurosmart. Smart USB Token. White Paper, Eurosmart," 2008. [Online].

[10] M. Bjørling, P. Bonnet, L. Bouganim and B. Þór Jónsson, "FLIP: Understanding the Energy Consumption of Flash Devices," *IEEE Data Engineering Bulletin,* vol. 33(4), pp. 48-54, 2010.

[11] T. Yan, D. Ganesan and R. Manmatha, "Distributed image search in camera sensor networks," *Sensys,* pp. 155-168, 2008.

[12] M. F. Porter, "An algorithm for suffix stripping," *Program 14,* 1980.

[13] J. Zobel, A. Moffat and K. Ramamohanarao, "Inverted files versus signature files for text indexing," *ACM Transactions on Database Systems (TODS),* vol. 23(4), pp. 453-490, 1998.

[14] U. MANBER and G. MYERS, "Suffix Arrays: A New Method for On-Line String Searches," *SIAM J. COMPUT,* vol. 22, p. 935–948, October 1993.

[15] I. H. Witten, A. Moffat and T. C. Bell, Managing Gigabytes: Compressing and Indexing Documents and Images, San Francisco, CA: 2nd Ed. Morgan Kaufmann, 1999.

[16] E. A. Fox and W. C. Lee, "FAST-INV: A fast algorithm for building large inverted files," Virginia Polytechnic Institute and State University , Blacksburg, VA, 1991.

[17] D. Harman and G. Candela, "Retrieving records from a gigabyte of text on a minicomputer using statistical ranking," *Journal of the American Society for Information Science,* vol. 41(8), p. 581–589, December 1990.

[18] W. Rogers, G. C and D. Harman, "Space and time improvements for indexing in information retrieval," *Proceedings of the Annual Symposium on Document Analysis and Information Retrieval,* 1995.

[19] A. Moff at and T. A. H. Bell, "In Situ Generation of Compressed Inverted Files," *Journal of the American Society for Information Science,* vol. 46(7), p. 537–550, 1995.

[20] N. Lester, J. Zobel and H. E. Williams, "In-Place versus Re-Build versus Re-Merge: Index Maintenance Strategies for Text Retrieval Systems," *ACSC '04 Proceedings of the 27th Australasian conference on Computer science,* vol. 26, pp. 15-23 , 2004.

[21] D. Cutting and J. Pedersen, "Optimizations for dynamic inverted index maintenance," *SIGIR '90 Proceedings of the 13th annual international ACM SIGIR conference on Research and development in information retrieval,* p. 405–411, 1990.

[22] A. Tomasic, H. García-Molina and K. Shoens, "Incremental updates of inverted lists for text document retrieval," *SIGMOD '94 Proceedings of the 1994 ACM SIGMOD international conference on Management of data,* vol. 23(2), pp. 289-300 , 1994 .

[23] G. M. Salton, A. Wong and C. Yang, "A vector space model for automatic indexing," *Communications of the ACM,* vol. 18(11), pp. 613-620 , Nov. 1975 .

[24] J. Zobel and A. Moffat, "Inverted files for text search engines," *ACM Computing Surveys (CSUR),* vol. 38(2), 2006.

[25] D. Salomon, Variable-length Codes for Data Compression, 2007.

[26] P. ELIAS, "Universal codeword sets and representations of the integers," *IEEE Trans. Inform. Theory,* p. 194–203, 1975.

[27] S. GOLOMB, "Run-length encodings," *IEEE Trans. Inform. Theory IT–12,* p. 399–401, 1966.

[28] A. SMEATON and C. J. VAN RIJSBERGEN, "The nearest neighbour problem in information retrieval," *Proceedings of the 4th Annual International ACM SIGIR Conference on Research and Development in,* p. 83–87, 1981.

[29] C. BUCKLEY and A. F. LEWIT, "Optimisation of inverted vector searches," *Proceedings of the 8th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval,* p. 97–110, 1985.

[30] A. MOFFAT and J. ZOBEL, "Self-indexing inverted files for fast text retrieval," *ACM Trans. Informa. Syst.,* p. 349–379, 1996.

[31] M. PERSIN, J. ZOBEL and R. SACKS-DAVIS, "Filtered document retrieval with frequency-sorted indexes," *J. Amer. Soc. Inform. Science 47,* vol. 10, p. 749–764, 1996.

[32] W. Y. P. WONG and D. K. LEE, "Implementations of partial document ranking using inverted files," *Inform. Proc. Manag.,* p. 647–669, 1993.

[33] M. PERSIN, J. ZOBEL and R. SACKS-DAVIS, "Filtered document retrieval with frequency-sorted indexes," *J. Amer. Soc. Inform. Science 47,* vol. 10, p. 749–7641, 1996.

[34] D. Ferraiolo, D. Kuhn and R. Chandramouli, Role-Based Access Control, Artech House, 2003.

[35] X. Zhang, F. Parisi-Presicce, R. Sandhu and J. Park, "Formal model and policy specification of usage control," *ACM Transactions on Information and System Security,* vol. 8(4), pp. 351-387, 2005.

[36] Yang, H. Barringer and N. Zhang, "A Purpose-Based Access Control Model," *Third International Symposium on Information Assurance and Security,* pp. 143-148, 2007.

[37] V. C. Hu, D. Ferraiolo, R. Kuhn, A. R. Friedman, A. J. Lang, M. M. Cogdell, A. Schnitzer, K. Sandlin, R. Miller and K. Scarfone, "Guide to attribute based access control (ABAC) definition and considerations (draft)," *NIST Special Publication,* 2013.

[38] T. L. Hinrichs, W. C. Garrison III, A. J. Lee, S. Saunders and J. C. Mitchel, "TBA : A Hybrid of Logic and Extensional Access Control Systems," *8th International Workshop, FAST 2011;Springer-Verlag Berlin Heidelberg,* vol. 7140, pp. 198-213, 2012.

[39] M. L. Mazurek, Y. Liang, W. Melicher, M. Sleeper, L. Bauer, G. R. Ganger, N. Gupta and M. K. Reiter, "Toward Strong, Usable Access Control for Shared," *FAST'14 Proceedings of the 12th USENIX conference on File and Storage Technologies,* pp. 89-103, 2014.

[40] M. Hart, C. Castille, R. Johnson and A. Stent, "Usable Privacy Controls for Blogs," *Computational Science and Engineering, 2009. CSE '09. International Conference on,* vol. 4, pp. 401 - 408, 29-31 Aug. 2009.

[41] C.-m. Au Yeung, L. Kagal, N. Gibbins and N. Shadbolt, "Providing Access Control to Online Photo Albums Based on Tags and Linked Data," *AAAI Spring Symposium on Social Semantic Web: Where Web 2.0 Meets Web 3,* pp. 23 - 25, 2009.

[42] P. Klemperer, Y. Liang, M. Mazurek, M. Sleeper, B. Ur, L. Bauer, L. F. Cranor, N. Gupta and M. Reiter, "Tag, you can see it!: using tags for access control in photo sharing," *CHI '12 Proceedings of the SIGCHI Conference on Human Factors in Computing Systems,* pp. 377-386, 2012.

[43] M. L. Mazurek, P. F. Klemperer, R. Shay, H. Takabi, L. Bauer and L. F. Cranor, "Exploring reactive access control," *CHI '11 Proceedings of the SIGCHI Conference on Human Factors in Computing Systems,* pp. 2085-2094, 2011.

[44] M. L. Mazurek, J. P. Arsenault, J. Bresee, N. Gupta, I. Ion, C. Johns, D. Lee, Y. Liang, J. Olsen, B. Salmon, R. Shay, K. Vaniea, L. Bauer, L. F. Cranor, G. R. Ganger and M. K. Reiter, "Access Control for Home Data Sharing: Attitudes, Needs and Practices," *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10),* pp. 645-654, 2010.

[45] E. Ioannou, J. De Coi, A. Koesling, D. Olmedilla and W. Nejdl, "Access Control for Sharing Semantic Data across Desktops," *CEUR Workshop Proceedings,* vol. 320, 2007.

[46] Q. Wang, H. Jin and N. Li, "Usable access control in collaborative environments: authorization based on people-tagging," *Proceedings of the 14th European conference on Research in computer security (ESORICS'09),* pp. 268-284, 2009.

[47] D. Agrawal, D. Ganesan, R. Sitaraman, Y. Diao and S. Singh, "Lazy-adaptive tree: An optimized index structure for flash devices.," *PVLDB,* vol. 2(1), pp. 361-372, 2009.

[48] Y. Li, B. He, R. Jun Yang, Q. Luo and K. Yi, "Tree indexing on solid state drives," *PVLDB,* Vols. 3 Issue 1-2, pp. 1195-1206, 2010.

[49] "http://www.memsic.com/userfiles/files/Datasheets/WSN/telosb_datasheet.pdf," [Online].

[50] Y. Diao, D. Ganesan, G. Mathur and P. Shenoy, "Rethinking data management for storage-centric sensor networks," *CIDR,* p. 22–31, 2007.

[51] N. Tsiftes and A. Dunkels, "A database in every sensor," *SenSys,* pp. 316-332, 2011.

[52] C.-H. Wu, T.-W. Kuo and L. P. Chang, "An efficient b-tree layer implementation for flash-memory storage systems," *ACM Transactions on Embedded Computing Systems (TECS),* vol. 6(3), 2007.

[53] B. Debnath, S. Sengupta and J. Li, "Skimpystash: Ram space skimpy key-value store on flash-based storage," *ACM SIGMOD International Conference on Management of Data,* p. 25–36, 2011.

[54] P. O'Neil, E. Cheng, D. Gawlick and E. O'Neil, "The Log-Structured Merge-Tree (LSM-Tree).," *Acta Informatica,* vol. 33(4), p. 351–385, 1996.

[55] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan and R. Kanneganti, "Incremental organization for data recording and warehousing," *PVLDB,* p. 16–25, 1997.

[56] C. Jermaine, A. Datta and E. Omiecinski, "A novel index supporting high volume data warehouse insertion," *PVLDB,* p. 235–246, 1999.

[57] C. Jermaine, E. Omiecinski and W. G. Yee, "The partitioned exponential file for database storage management," *VLDBJ,* vol. 16(4), pp. 417-437, 2007.

[58] R. Sears and R. Ramakrishnan, "bLSM: a general purpose log structured merge tree," *SIGMOD,* pp. 217-228, 2012.

[59] D. Bell and L. LaPadula, "Secure computer systems: Unified exposition and multics interpretation (Technical Report ESD-TR-73-306)," The MITRE Corporation, 1976.

[60] M. A. Harrison, W. L. Ruzzo and J. D. Ullman, "Protection in Operating Systems," *Communication of the ACM,* 1976.

[61] N. Anciaux, S. Lallali, I. Sandu Popa and P. Pucheral:, "A Scalable Search Engine for Mass Storage Smart Objects," *PVLDB,* vol. 8(9), pp. 910-921, 2015.

[62] N. Anciaux, L. Bouganim, Y. Guo, P. Pucheral, J.-J. Vandewalle and S. Yin, "Pluggable personal data servers," *SIGMOD,* pp. 1235-1238, 2010.

[63] S. LALLALI, A. Anciaux, I. Sandu Popa and P. Pucheral, "A Secure Search Engine for the Personal Cloud," *SIGMOD Conference,* pp. 1445-1450 , 2015.

[64] "INRIA, LIRIS, UVSQ, GEMALTO, CryptoExperts, CG78. 2012. Use cases and functional architecture specification, KISS deliverable ANR-11-INSE-0005-D1, 21/12/2012.".

[65] J. Kim and W. B. Croft, "Retrieval Experiments using Pseudo-Desktop Collections," *CIKM,* pp. 1297-1306, 2009.

[66] L. Kissner and D. Song, "Privacy-Preserving Set Operations," *CRYPTO ,* 2005.

[67] D. Kempe, A. Dobra and J. Gehrke, "Gossip-based computation of aggregate information," *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science (FOCS'03) ,* pp. 482 - 491, 2003.

[68] Q.-C. To, B. Nguyen and P. Pucheral, "Privacy-Preserving Query Execution using a Decentralized Architecture and Tamper Resistant Hardware," *EDBT,* pp. 487-498, 2014.

[69] T. Allard, N. Anciaux, L. Bouganim, Y. Guo, L. Le Folgoc, B. Nguyen, P. Pucheral, I. Ray, and S. Yin, "Personal Data Servers: a Vision Paper," *Proceedings of the VLDB Endowment,* Vols. 3(1-2), pp. 25-35 , 2010.